Interfacing algebraic and numeric computation

Dewar, Michael C.

Award date:
1991

Awarding institution:
University of Bath

INTERFACING ALGEBRAIC AND NUMERIC COMPUTATION

submitted by

Michael C. Dewar

for the degree of Ph.D

of the

University of Bath

1991

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Summary

There are two different approaches used to solve mathematical problems with computers. The more traditional numerical approach is characterised by libraries of FORTRAN subprograms. These tend to be hard to use, not least because they require the user to program in FORTRAN. The other approach is to apply symbolic techniques, usually through a computer algebra system. Both these methods have their merits in particular situations and with different problems. This thesis deals with the design and implementation of tools to enable a user to apply both approaches to solving problems within the same environment.

Acknowledgements.

I would like to express my thanks to NAG Ltd. and its associates for their active cooperation and participation during this project, in particular Mike Richardson for all his hard work designing routine interfaces and test examples. I'd also like to thank Tony Hearn the author of Reduce, and Barbara Gates the author of GENTRAN, whose work provided the basis on which IRENA was built. Finally I'd like to thank my supervisor, Professor James Davenport, for all his advice and encouragement.

This work was funded by the Science and Engineering Research Council of the United Kingdom through a research studentship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since they were invented, one of the main uses of electronic computers has been the solution of problems in mathematics. In the early days the lack of sophistication of existing hardware forced programmers to go to great lengths to produce software capable of solving difficult problems. The legacy of these pioneering efforts is still with us today in the structure of languages like FORTRAN, and the packaging of algorithms into libraries of subprograms. Such systems compromise accessibility and usability in favour of power and flexibility — whilst adaptable and modular, they can only be exploited via complex programs written in fairly low-level languages. Given the capabilities of modern hardware, such a compromise is to a large extent no longer necessary.

Modern computer users have a diverse variety of backgrounds. They may be experts in their own field, but know little or nothing about computer programming. They prefer to use systems tailored for their own requirements: systems which accept a representation of the problem to be solved in a form which is natural for the user; and return their results in the same style. Such users want to exploit the power of computers and the various packages which exist for them, without the inconvenience of having to program in low-level languages like FORTRAN or C. Their ideal language is a language which represents the ideas they have about the problems which they want to solve.

1.1 Numerical Libraries

As computers became more widespread and accessible to ordinary users, it became clear that much effort was being duplicated as individuals wrote their own implementations of well-known algorithms for numerical analysis, statistics, data sorting and so on. This led to the concept of a *subprogram library* which contained high-quality implementations of various algorithms which the user could link into his or her program during compilation. Although libraries were produced in a wide variety of languages, the most common was FORTRAN.

The advantages of subprogram libraries are clear: they provide efficient, reliable, and thoroughly tested pieces of code. However there are disadvantages as well. Although FORTRAN is fast, it is a very unintuitive language. Even for an experienced programmer, it can take some time to write and test the code to solve a relatively trivial problem. Not only does he or she need to translate it from its mathematical definition to its FORTRAN one, the user must frame it in the way required by the particular routine. Many conceptually simple operations are error-prone when done by hand: for example writing a subroutine to return the jacobian of a given set of equations in an array. FORTRAN syntax is very low-level when compared with that of modern programming languages, and the restriction of parameter names to six characters prevents them from being either meaningful or memorable.

Another problem with FORTRAN is the amount of "unnecessary" information which the user must provide. Arrays for workspace, whose size depends in some way on the particular problem being solved, and array dimensions must all be explicitly passed as parameters to the routine. Because of its call-by-reference semantics the names of all parameters used to output results must also be passed. The correct ordering of the parameters is imperative, and since routines typically take not less than a dozen parameters (and often thirty or more) it is in practice impossible to use libraries without access to fairly detailed documentation, either printed or online.

Another aspect of using these "canned" algorithms is that a user needs a little understanding, not only of the problem being solved, but also of the method being used to solve it. This is because the routine will typically require a number of parameters

to control the operation and termination of the algorithm: the maximum number of iterations to take in a quadrature routine, the step size to be used in finding the zero of a polynomial, or the accuracy required for the solution of a differential equation. The latter example also requires some knowledge of the precision of the implementation being used, and how well the algorithm may be expected to perform under the circumstances.

There is also the problem of choosing which algorithm to use in the first place, and this may involve some higher level of mathematical knowledge. For example a user wishing to solve a set of differential equations may have to decide whether it is stiff or not, while someone using a quadrature routine might need to determine whether the integrand has any singularities or discontinuities. Advice giving systems, whether printed decision trees or interactive menu-driven systems, all tend to expect the user to understand such concepts, and be able to apply them to the particular problem in hand.

1.1.1 The NAG FORTRAN Library

The example studied in this thesis is concerned with a particular library: the NAG FORTRAN Library [NAG LTD. 1990]. The Library was started in 1970 and has grown steadily ever since. It is divided into chapters, each of which contains routines for solving particular problems (for example optimisation, quadrature, time series analysis etc.). Most chapters have a three-character name based on the conventions adopted for publishing numerical algorithms by the ACM, and each routine in a chapter suffixes this with a unique three-character label. So for example chapter A02 is concerned with complex arithmetic, and contains the three routines A02AAF, A02ABF, A02ACF. A new mark is released roughly every 18 months, which contains some new routines and has had other older ones deleted. Thus some routines may be 15 years old, while others were written very recently. A consequence of this is that many routines were written in FORTRAN-IV at a time when the efficient use of memory was a far more critical consideration than in modern computing environments. This is reflected in the way data is handled and packed, and the way arrays are often used for multiple purposes in different situations. The resulting user interface is often extremely confusing and

unintuitive.

At mark 14 (March 1990) the Library contained around 1000 user-callable routines. The routines are contributed by academics and commercial institutions throughout the world, which leads to a great deal of inconsistency in the interfaces, particularly in the naming and style of parameters (i.e. a subprogram argument which fulfills the same rôle in two subprograms may be of a completely different — and hence incompatible — format in two routines which solve the same problem).

The only fairly consistent part of the interfaces is the way in which they handle errors. There is a parameter, IFAIL, which the user sets to either -1, 0, or 1 before entering the routine. This determines whether, on encountering a fatal error, the routine will:

- hand control back to the calling program with a printed message (noisy soft fail);
- terminate the program with a printed message (hard fail);
- hand control back to the calling program without a message (silent soft fail).

If one of the soft fail options is chosen then, on exit, the value of IFAIL will have been set to an integer value which indicates what sort of error has occurred. The interpretation of these values is given in the printed and online documentation. Thus when the soft fail option is chosen it is always vital for the calling program to check the value of IFAIL on exit from a Library routine.

The NAG Library is implemented on a wide range of machines, and has established an international reputation for the excellence of the quality of its algorithms. However it is only of immediate appeal to people who enjoy programming in FORTRAN, an increasingly rare breed! Todays computer *user* does not necessarily expect to be a computer *programmer*, but expects to be able to use interactive, attractive systems which "understand" the problem *in the same terms as he or she does*. If that fails, then the user can enlist the services of an expert. Yet the Library represents an enormous investment of time and expertise, and to abandon it would be an appalling waste of manpower and resources. The system described later in this thesis offers an alternative path, adapting the existing library to modern expectations.

1.2 Computer Algebra Systems

Computer Algebra systems manipulate symbols not numbers. Rather than using the approximation methods of numerical analysis, they use exact algebraic techniques. Such systems tend to be interactive programs, commonly written in some dialect of LISP, and they accept their input in a quasi-mathematical notation which is simple to use and remember. They can give general expressions as an answer (to an integral, for example), rather than only a numerical value. All the details of the algorithms used are normally hidden from the user, and the statement of the problem consists only of its logical, mathematical components.

Unfortunately computer algebra systems can be comparatively slow, and the time taken to solve a given problem is unpredictable, as is the size of the solution. For some problems, such as solving ordinary differential equations, they may not offer a "black box" interface but rather expect the user to explicitly perform the steps required to find a solution, using the system to perform the manipulation.

Computer Algebra systems normally deal with the rational numbers and their polynomial extensions. There are generally no limits to the size of integers which the system can handle, and so any rational number can be represented exactly. Computer Algebra systems also offer floating point arithmetic and, because the operations can be carried out in software rather than in hardware, they can offer arbitrarily high precision, specified at runtime. Some systems use both hardware and software floating point, either explicitly as in REDUCE 3.3, or transparently as in REDUCE 3.4. Although considerably slower than hardware floating point, arbitrary precision *bigfloats* offer obvious advantages. However, no matter how high a precision is chosen, rounding errors can never be eliminated altogether. Some algebraic techniques, such as Buchberger's algorithm for determining the Gröbner basis of a set of polynomials, are extremely sensitive to this and so, where possible, it is better to work in the rational field.

Numerical algorithms implemented in computer algebra systems and using floating point will generally run far slower than those implemented in FORTRAN libraries. It is perfectly possible to implement these algorithms in such a way as to use exact rational arithmetic but this leads to two problems:

- Comparing the relative size of two rational numbers whose denominators may be extremely large integers is not easy.
- Trigonometric and logarithmic functions normally only yield values in floating point. While their exact value can by expressed as a convergent power series, truncating this to yield a rational approximation introduces rounding error and again leads to the difficulty of comparing two rational numbers.

The problem of comparing two rational numbers has been extensively treated by [Kornerup & Matula 1979], however the algorithms proposed are still relatively slow. It makes little sense in most cases to try and use exact arithmetic since we are only using approximation techniques anyway.

1.2.1 The Reduce Computer Algebra System

For the rest of this thesis we shall be mainly concerned with one computer algebra system: Reduce [Fitch 1985, Hearn 1987]. This system has been under development since the late 1960s, and is available on a wide range of machines. It has been implemented in many dialects of LISP, most notably PSL [Galway *et al.* 1987] which was specifically designed as a Reduce platform. It offers sophisticated algorithms for such tasks as indefinite integration, polynomial equation solving, and matrix manipulation. All expressions are represented as quotients of polynomials, though they may contain terms involving the trigonometric and logarithmic functions, as well as functions defined by the user. There are facilities for substituting values into expressions, extracting their coefficients, and performing all the normal arithmetic operations on them. Although output is normally a "three-line" mathematical style, it can be generated as REDUCE input syntax, FORTRAN expressions, or even TEX notation[1].

REDUCE offers two *modes* of operation: *algebraic* and *symbolic*. System program is written in the latter, which is essentially a pre-processor with algol-like syntax for the underlying LISP system, while applications programs are written in the former. Although REDUCE is untyped, it has the concept of a *domain* [Bradford *et al.* 1986].

[1] Using one of the packages TRI or RLFI from the Network Library.

The domain can be set by the user, and currently includes rational, rounded, Gaussian integer, complex rational, and complex rounded. Operations are performed in the current domain, and constants from other domains coerced to this when necessary. The rounded domain consists of floating point numbers. Where the precision is small enough these are represented as hardware floats, otherwise they are bigfloats [Sasaki 1979].

REDUCE offers sophisticated packages for generating and optimising code in FORTRAN, RATFOR, C and PASCAL, which are discussed in detail in section 2.

1.3 NAG versus Reduce

Reduce offers the interactive environment with comprehensible syntax and uncluttered interface which modern computer users want. It allows them to concentrate on the *mathematics* of a problem without bothering about the underlying algorithms, or the dubious pleasures of FORTRAN programming. However like all computer algebra systems it is not an environment suited to numerical algorithms for the reasons given in section 1.2. Even if this was not so, the effort required to implement even a significant subset of (say) the NAG Library's capabilities under Reduce would be enormous. Thus a user is often going to be faced with the choice: which method to use. In some situations the choice is clear, since only one system will be capable of solving a particular problem. However often both systems will appear to be capable of offering a solution.

Let us consider the problem of differentiation. It is well known that numerical differentiation is extremely sensitive to rounding error, whereas symbolic differentiation is an almost trivial problem. This is why for many applications (optimisation, solving differential equations etc.) numerical routines require not only the a piece of code to represent a function, but another to represent its derivative(s) as well.

On the other hand, consider the problem of definite integration. Here the pros and cons are not so clear cut. Reduce has an algebraic integration package which can handle a wide range of integrals, returning a function which can be evaluated at the end points of the range of integration to yield the result in the usual way (provided, of course, that the integral has no singularities in the range). NAG provide various routines, some of

them general purpose, and others designed to deal with a particular class of integral. Again, the time taken to prepare the problem for solution is much less in Reduce, but it may take a long time to yield an answer, and quite often fail altogether. However, if the same integral is going to be solved over different regions, then numerical evaluation of a symbolic solution may well be faster.

There are some problems which Reduce can't handle at all — for example solving non-factorable algebraic equations of degree five or more — which NAG routines will often find trivially easy.

In many cases Reduce scores over NAG because of its friendly and easy-to-use user interface. Most users with the choice will try using Reduce first, before going to the effort of writing even small FORTRAN programs. This increased execution time is often more than compensated for by the decreased *formulation* time, the time taken to present the user's problem in a form comprehensible to the mathematical software. A nicer interface to the NAG library would encourage users to think more carefully about their choice of methods.

1.4 Combining Symbolic and Numeric methods

Some problems benefit from a combination of both the symbolic and numerical approach. For example it is often convenient to model a problem symbolically, deriving a family of equations which describe it, and then evaluate those equations with various numerical values. There are several surveys of such applications [Fitch 1979, Ng 1979, Fitch 1990], most of which are concerned with problems in physics and mechanics. Recently there has also been some interest from the field of biochemistry to model enzyme reactions [Bennett *et al.* 1988, Fisher 1990a, Fisher 1990b].

The point here is that the two approaches — symbolic and numeric — should not be viewed as divorced from one another, but rather as complementary tools for problem solving. We discuss the technology which currently exists for harnessing the two methodologies in Chapter 2.

1.5 Summary

Existing libraries of FORTRAN subprograms represent decades of effort, and have proved themselves to be robust and reliable. Unfortunately they are difficult to use, and do not fit comfortably into the modern interactive, package-oriented, view of computer software. Computer algebra systems are friendly, interactive packages which, with the growth of computer power available to most users, are deservedly becoming more popular.

The two approaches are useful in different circumstances, but can be combined very effectively to tackle a number of classes of problems. Clearly any problem-solving toolkit for the modern scientist should include both symbolic and numerical facilities. It would be nice if such a toolkit could exploit the investment of effort and ability which a conventional subprogram library represents, while offering the user a more up-to-date interface.

Chapter 2

Related Work

In this chapter we will describe the previous work which has been done on interfacing symbolic and numerical computation, and the facilities which already exist for those wishing to mix the two paradigms.

2.1 Primitive FORTRAN generation

Most computer algebra systems have some rudimentary method for producing output in FORTRAN. For example, in Reduce we can set the switch FORT and all output will be converted to FORTRAN-compatible syntax. Reduce's system is comparatively sophisticated: most FORTRAN compilers will only accept a certain number of continuation lines, so it will *segment* any expression which is too large, and split it into several statements. The maximum acceptable number of continuation lines and their width are under user control, and default to the definitions in the ANSI standard [ANSI 1978]. A simple example is given in figure 2-1. Many computer algebra systems do not offer segmentation, and as such are very often useless, since the sort of expressions which are generated (for example jacobians) are normally very large indeed. This is the situation in Mathematica and Maple IV.

The user also has to think carefully about how the computer algebra system is going to "simplify" the expressions being translated. In the example in figure 2-1 it is clearly far more efficient to prevent Reduce expanding the right hand side of the assignment,

```
REDUCE Development Version, 22-Jan-90 ...

1: on fort;

2: cardno!* := 2$ % Limit the number of continuation lines to 1

6: f := (x+y)^10;
      ANS2=10.*X*Y**9+Y**10
      ANS1=252.*X**5*Y**5+210.*X**4*Y**6+120.*X**3*Y**7+
     . 45.*X**2*Y**8+ANS2
      F=X**10+10.*X**9*Y+45.*X**8*Y**2+120.*X**7*Y**3+210.
     . *X**6*Y**4+ANS1
```

Figure 2-1: An example of simple FORTRAN generation in Reduce.

which can be done by setting the Reduce switch EXP off. This issue, how to produce efficient code, will be discussed in section 2.4.

2.2 GENTRAN

Reduce offers even more sophisticated code-generation facilities, however, with the GENTRAN package [Gates 1985, Gates 1986, Gates 1987]. This allows the user to generate complete programs in FORTRAN, RATFOR, PASCAL or C, rather than just isolated expressions. Not only can GENTRAN translate most LISP statements, but the user may build skeletal programs, or *templates*, which are then "fleshed-out" by Reduce. The *passive* parts of the template contain fragments of code in the target language, and are echoed verbatim to the output, while the *active* parts consist of sequences of Reduce or GENTRAN commands. GENTRAN has a separate file-handling system, and so output from its functions may be redirected to selected files, while the results of Reduce functions still appear on the screen. There are also facilities for handling type declarations, and segmentation. In the latter case automatically-generated temporary variables are automatically declared to be of a default type (usually REAL).

Other facilities which we have added to the original version of GENTRAN, including the handling of double precision constants and variables, the coercion of the arguments of FORTRAN intrinsic functions to the correct type, and the generation of

complex constants, are given in appendix B. The GENTRAN package was originally implemented in Macsyma [Gates & Wang 1984].

As GENTRAN is central to the work described later in this thesis, we shall give a simple example which gives a flavour of some of its facilities. Suppose that we wish to generate a FORTRAN function which will return either a polynomial evaluated at a point, or its partial derivative. In figure 2-2 we see a GENTRAN template to perform this task. The `;BEGIN; ... ;END;` sequences enclose the active parts. Since we cannot tell in advance whether GENTRAN will generate any extra variables (to reduce expressions to the size allowed by the FORTRAN compiler), we produce the final program segment in two phases. Our template creates a second GENTRAN template whose only active part generates the symbol table. This template, shown in figure 2-3, is then processed to get the final FORTRAN code. The sequence of steps needed in Reduce to do this is shown in figure 2-5, and the final result in figure 2-4. In practice, this multiple-pass technique is almost always necessary when translating expressions into FORTRAN.

2.3 Other Code Generation Packages

There are a number of other systems under development to provide complete high-quality code and program generation facilities for computer algebra systems.

GENCRAY [Weerawarana & Wang 1989] is a package which can produce either FORTRAN 77 or Cray FORTRAN from Vaxima (the VAX version of Macsyma). It is interesting for two reasons. The first is that, unlike other systems, it doesn't produce its output directly from LISP; but rather produces an intermediate form which can subsequently be transformed by a C program. This is built using standard parsing techniques like *yacc* and *lex*, the idea being to make the system more easily extensible. In addition, by isolating the system dependent part of the parsing process it is also supposed to be more portable. The other novel feature of GENCRAY is that it includes facilities for the generation of parallel and vectorisable code for the Cray. This is done through a suite of macros which handle matrix and vector operations; and a second set which allows the user to specify that a set of procedure calls be done in parallel, or set up pipelining etc.

```
;BEGIN;
     % A Gentran Template to generate a double precision function to
     % return the value of a multivariate polynomial or its partial
     % derivative at a given point.

     off gendecs$ % Postpone the generation of type declarations.
     on double$ % Force double precision output.
     tempvartype!* := 'real$ % Temporary variables will have type real.
;END;
     DOUBLE PRECISION FUNCTION F(X,Y,Z,FVALS)
;BEGIN;
     gentran declare <<f : function;
                       x,y,z : real;
                       fvals : logical;
                     >>$

     gentran literal ";BEGIN;",cr!*,tab!*,"on gendecs$",cr!*,";END;",
                     cr!*$
;END;
     IF (FVALS) THEN
;BEGIN;
     gentran f :=: f$
;END;
     ELSE
;BEGIN;
     gentran f :=: df(f,x)$
;END;
     ENDIF
     RETURN
     END
;BEGIN;
     gentran literal "END;",cr!*$
;END;
```

Figure 2-2: A Gentran Template.

```
     DOUBLE PRECISION FUNCTION F(X,Y,Z,FVALS)
;BEGIN;
     on gendecs$
;END;
     IF (FVALS) THEN
     F=X**5+5.0D0*X**4*Y+5.0D0*X**4*Z+10.0D0*X**3*Y**2+20.0D0*X**3*Y*Z+
    . 10.0D0*X**3*Z**2+10.0D0*X**2*Y**3+30.0D0*X**2*Y**2*Z+30.0D0*X**2*
    . Y*Z**2+10.0D0*X**2*Z**3+5.0D0*X*Y**4+20.0D0*X*Y**3*Z+30.0D0*X*Y**
    . 2*Z**2+20.0D0*X*Y*Z**3+5.0D0*X*Z**4+Y**5+5.0D0*Y**4*Z+10.0D0*Y**3
    . *Z**2+10.0D0*Y**2*Z**3+5.0D0*Y*Z**4+Z**5
     ELSE
     F=5.0D0*X**4+20.0D0*X**3*Y+20.0D0*X**3*Z+30.0D0*X**2*Y**2+60.0D0*X
    . **2*Y*Z+30.0D0*X**2*Z**2+20.0D0*X*Y**3+60.0D0*X*Y**2*Z+60.0D0*X*Y
    . *Z**2+20.0D0*X*Z**3+5.0D0*Y**4+20.0D0*Y**3*Z+30.0D0*Y**2*Z**2+
    . 20.0D0*Y*Z**3+5.0D0*Z**4
     ENDIF
     RETURN
     END
;END;
```

Figure 2-3: The intermediate template.

```
     DOUBLE PRECISION FUNCTION F(X,Y,Z,FVALS)
     DOUBLE PRECISION X,Y,Z
     LOGICAL FVALS
     IF (FVALS) THEN
     F=X**5+5.0D0*X**4*Y+5.0D0*X**4*Z+10.0D0*X**3*Y**2+20.0D0*X**3*Y*Z+
    . 10.0D0*X**3*Z**2+10.0D0*X**2*Y**3+30.0D0*X**2*Y**2*Z+30.0D0*X**2*
    . Y*Z**2+10.0D0*X**2*Z**3+5.0D0*X*Y**4+20.0D0*X*Y**3*Z+30.0D0*X*Y**
    . 2*Z**2+20.0D0*X*Y*Z**3+5.0D0*X*Z**4+Y**5+5.0D0*Y**4*Z+10.0D0*Y**3
    . *Z**2+10.0D0*Y**2*Z**3+5.0D0*Y*Z**4+Z**5
     ELSE
     F=5.0D0*X**4+20.0D0*X**3*Y+20.0D0*X**3*Z+30.0D0*X**2*Y**2+60.0D0*X
    . **2*Y*Z+30.0D0*X**2*Z**2+20.0D0*X*Y**3+60.0D0*X*Y**2*Z+60.0D0*X*Y
    . *Z**2+20.0D0*X*Z**3+5.0D0*Y**4+20.0D0*Y**3*Z+30.0D0*Y**2*Z**2+
    . 20.0D0*Y*Z**3+5.0D0*Z**4
     ENDIF
     RETURN
     END
```

Figure 2-4: An example of FORTRAN generated by Gentran.

```
REDUCE Development Version, 10-Jul-90 ...

1: f := (x+y+z)^5$

2: gentranin "foo.tem" out "#foo.tem"$

"#foo.tem"

21: gentranin "#foo.tem" out "foo.f"$

"foo.f"
```

Figure 2-5: A Reduce session using Gentran.

MathCode [Kant *et al.* 1990] is a package for Mathematica which allows the user to produce FORTRAN or C programs. Originally designed as part of a package to generate programs for mathematical modelling, it consists of a set of commands to generate particular constructs in the target language. Most of these commands are the lower case equivalents of the equivalent Mathematica functions. The constructs may be subprogram headers, loops, assignments etc. It has a similar facility to GENTRAN's templates: the *Splice* command will read a FORTRAN or C program containing embedded Mathematica commands and on output replace these with their evaluated results. It does not appear to segment large expressions or support double precision output.

MACROFORT [Gomez 1990] is a FORTRAN code generator for MAPLE. Like MathCode, it has a suite of functions for generating particular program constructs. In this case they have the same name as the target FORTRAN construct suffixed with either an "f" or "m". Thus the user needs to know a little bit about FORTRAN syntax. A useful feature is that translation is deferred until after each program unit or subprogram is completely defined, so type declarations, `COMMON` blocks etc. may be specified at any time but will appear at the right point in the generated code. It has an optimiser (see § 2.4) and will produce single or double precision output.

2.4 Code Optimisation

When used correctly, GENTRAN is a very powerful tool. However it is possible for the naïve user to produce very inefficient code. In theory this shouldn't matter, since most FORTRAN compilers have the facility to optimise arithmetic expressions. However there are several problems with this:

- Large numbers of large expressions can still outstretch the capacity of a given compiler.
- Exponentiation is often treated as a function call, so the obvious relationship between e.g. x^2 and x^4 is not recognised.
- Optimising compilers do not assume the validity of the commutative and distributive laws, and so do not produce optimal expressions. This is partly because, in principle, these laws do not hold with floating point numbers though, in practice, empirical evidence suggests that this problem can be ignored [van Hulzen 1984].

There has been a great deal of work done in this area. The most common approach is to remove common sub-expressions from the generated code, thus reducing the number of arithmetic operations at the expense of an increase in the number of assignments. This approach doesn't take account of the relative costs of the different operations, which will vary from architecture to architecture, but is completely portable. A simple example would be converting the expansion of $(x + y)^{10}$ from:

```
F=X**10+10.0*X**9*Y+45.0*X**8*Y**2+120.0*X**7*Y**3+
. 210.0*X**6*Y**4+252.0*X**5*Y**5+210.0*X**4*Y**6+120.0
. *X**3*Y**7+45.0*X**2*Y**8+10.0*X*Y**9+Y**10
```

to the sequence:

```
T0=X*Y
T5=X**2
T10=Y**2
```

```
      T12=T0**2
      T13=T0*T12
      T15=T5**2
      T17=T10**2
      T18=T15*T5
      T19=T17*T10
      F=T12*(252.0*T13+45.0*(T18+T19))+T12**2*(210.0*(T10+T5
     . ))+T13*(120.0*(T17+T15))+T0*(10.0*T17**2+10.0*T15**2)
     . +T15*T18+T17*T19
```

which reduces the number of multiplications from eighteen to sixteen and the number of exponentiations from eighteen to eight, at the expense of nine extra assignments.

There are other optimisation strategies as well, in particular rewriting polynomials in a nested form, replacing all the exponentiations by multiplications. Doing this here will produce the expression:

```
      F=Y**10*(1.0+(X/Y)*(10.0+(X/Y)*(45.0+(X/Y)*(120.0+(X/Y
     . )*(210.0+(X/Y)*(252.0+(X/Y)*(210.0+(X/Y)*(120.0+(X/Y)
     . *(45.0+(X/Y)*(10.0+X/Y)))))))))
```

which, after we remove the sub-expression `X/Y`, consists of only ten multiplications, ten additions, one division, one exponentiation and two assignments. This is a generalisation of the famous Horner's rule which replaces the polynomial:

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

by the expression:

$$(\ldots(a_n x + a_{n-1})\, x + \ldots)\, x + a_0$$

Although this can be the most efficient method in many cases it is less general than common sub-expression removal, and can get tricky where we are dealing with large numbers of variables.

Symbolic code optimisation really comes into its own when computing jacobians. Because of the obvious relationships between the expressions, their density can be reduced and the efficiency of the resulting code dramatically improved by this technique. For example, optimising the jacobian of the following functions [Dewar 1989]:

$$
\begin{aligned}
f1(x1,x2,x3) &= \frac{x1+1.0}{15*x2+1*x3-0.14} \\
f2(x1,x2,x3) &= \frac{x1+2.0}{14*x2+2*x3-0.18} \\
f3(x1,x2,x3) &= \frac{x1+3.0}{13*x2+3*x3-0.22} \\
f4(x1,x2,x3) &= \frac{x1+4.0}{12*x2+4*x3-0.25} \\
f5(x1,x2,x3) &= \frac{x1+5.0}{11*x2+5*x3-0.29} \\
f6(x1,x2,x3) &= \frac{x1+6.0}{10*x2+6*x3-0.32} \\
f7(x1,x2,x3) &= \frac{x1+7.0}{9*x2+7*x3-0.35} \\
f8(x1,x2,x3) &= \frac{x1+8.0}{8*x2+8*x3-0.39} \\
f9(x1,x2,x3) &= \frac{x1+9.0}{7*x2+7*x3-0.37} \\
f10(x1,x2,x3) &= \frac{x1+10.0}{6*x2+6*x3-0.58} \\
f11(x1,x2,x3) &= \frac{x1+11.0}{5*x2+5*x3-0.73} \\
f12(x1,x2,x3) &= \frac{x1+12.0}{4*x2+4*x3-0.96} \\
f13(x1,x2,x3) &= \frac{x1+13.0}{3*x2+3*x3-1.34} \\
f14(x1,x2,x3) &= \frac{x1+14.0}{2*x2+2*x3-2.10} \\
f15(x1,x2,x3) &= \frac{x1+15.0}{1*x2+1*x3-4.39}
\end{aligned}
$$

gives the following improvements:

	+,-	*,/	**	=
Unoptimised	75	124	45	60
Optimised	38	93	30	136

2.4.1 SCOPE

SCOPE [van Hulzen *et al.* 1989] implements the algorithms to search for common sub-expressions in blocks of straight-line code. Its input consists of sets of Reduce expressions, and its output may be either Reduce expressions, or (through Gentran) FORTRAN or C. Its interface also allows the user access to the Gentran symbol table, and to control exactly when expressions are evaluated. A short example is given in figure 2-6.

```
33: f := (x+2y-z)^5$

34: optimize {f1:=:df(f,x),f2:=:df(f,y),f3:=:df(f,z)}
34:          declare <<x,y,z:real>>;

    REAL X,Y,Z,G45,G46,G47,G65,G64,G63,G72,G73,F1,F2,F3
    G45=Z*Y
    G46=Z*X
    G47=Y*X
    G65=Z*Z
    G64=Y*Y
    G63=X*X
    G72=G65*G65+16*G64*G64-(48*G64*G46)+6*G46*G46+G63*G63+24*(G45*G45+
    . G65*G47-(G63*G45)+G47*G47)+32*(G64*G47-(G64*G45))+8*(G63*G47-(G65
    . *G45))+4*(-(G65*G46)-(G63*G46))
    G73=5*G72
    F1=G73
    F2=10*G72
    F3=-G73
```

Figure 2-6: An example of the operation of the SCOPE package.

The package also contains commands to allow the user to use the obviously visible structure of an expression (similar to the Reduce *structr* operator), or to apply Horner rules.

There are unfortunately a few drawbacks with SCOPE. The most serious of these is that the current version cannot handle polynomials whose coefficients are domain elements, in particular floating point numbers. It contains a type inference mechanism to determine the type of the new variables it creates, but this is imperfect and tends

to give over-general results (i.e. the sum of two integer variables will be declared as a real).

2.4.2 Compress

We have already seen that Reduce expands expressions out fully, whereas more efficient code may result if this expansion is not carried out. One method of doing this is to turn the switches MCD and EXP off, which stops expressions being put over a common denominator and expanded respectively. Unfortunately this is a somewhat unrealistic approach, since other algebraic processes which a user might want to employ to generate the expressions to be translated may not, as a result, work properly.

An better method is that implemented by the COMPRESS package [Hulshof 1983]. This seeks to restore some or all of the structure to an expression, and may be used as a pre-processor to another optimiser.

2.4.3 Using Higher-Level Knowledge

A number of people have advocated employing knowledge about the process being used to generate the expressions to improve the efficiency of the final code. Although not a very general technique, this can lead to extremely good results. [Wang 1985] exploited the symmetries of his problems in finite element analysis to reduce the complexity of the FORTRAN he was generating using GENTRAN. [Mutrie *et al.* 1987] advocated a rather more ambitious approach in Maple whereby knowledge about the algorithms used to perform the "basic" algebraic operations such as differentiation, taylor series approximation, and integration, would be used to produce more efficient expressions.

2.4.4 Summary

It is clear that, for a given problem, a little careful thought may dramatically improve the efficiency of the code being generated. In general, however, most users will simply try to eliminate common sub-expressions using SCOPE or a similar package. In cases where expressions have a basic structure which has been "lost" through some algebraic expansion however, trying to restore that structure will often give more efficient code:

evaluating $(x+y)^{10}$ is more efficient than evaluating the expanded then optimised version given earlier.

2.5 Systems for solving specific problems.

Various systems have been built to solve problems in a specific area. In general the idea is for the program to take a description of the user's problem in symbolic form, analyse and manipulate it symbolically and then output a piece of "ready-to-run" FORTRAN code. This program can then be compiled and run to get a numerical solution to the problem.

[Barton *et al.* 1971] describe a program to solve differential equations via the method of Taylor series. The user interface accepts a description of the problem in a quasi-English syntax which is parsed and interpreted to produce a matrix representation of the problem. This matrix is then optimised to remove common sub-expressions and eliminate unnecessary operations between taylor series. Finally a FORTRAN program is generated to solve the problem.

FINGER [Wang 1986] is a package to generate code for finite element analysis which runs under MACSYMA and GENTRAN. The code it produces is designed to be linked to a particular library of FORTRAN subroutines. Another package concerned with finite element methods is that of [Barbier *et al.*].

2.6 NAGLINK

NAGLINK [Broughan 1986, Broughan 1987] was the first attempt to produce a comprehensive system unifying both symbolic and numerical methods in one computing environment. It used MACSYMA for the symbolic features and the NAG Library for the numerical algorithms. Each NAG routine had a corresponding LISP routine which translated the user-provided parameters into the required FORTRAN program. Parameters which were themselves subprograms had to be seperately converted into FORTRAN. The resulting program was then compiled, linked, and finally executed under MACSYMA, the results being returned as a list. NAG parameters which were

not strictly part of the problem specification, such as tolerances on results or limits on the number of iterations, were given global values which the user could change at need. Unfortunately the system never really became available outside the University of Waikato where it was written, because by the time it was finished some of the MACSYMA features used to execute the compiled code were no longer supported in the latest version.

In addition, there were some criticisms of the design of the interfaces. There were inconsistencies between the parameters to similar routines which, while to some extent reflecting the same faults in the NAG Library itself, were unnecessary. A significant number of routines were reported as not functioning correctly. This may partly have been due to the way in which each interface was hand-written, which also made upgrading the system with new releases of the Library a long and tedious job.

Since MACSYMA was no longer a suitable platform the authors decided to implement their own stand-alone LISP front-end to the system. Although primarily an interpreter for the NAG interfaces, SENAC [Punjani & Broughan 1990] does include implementations of a small number of the better known algorithms normally found in computer algebra systems, such as the Risch algorithm for closed-form integration [Schou & Broughan 1989]. This system was released commercially in late 1990, and appears very similar to its predecessor. In addition, it also offers a graphical facility via an interface to the NAG Graphics Library.

Chapter 3

IRENA

We have stated that there is a strong need for a unified environment in which to employ both symbolic and numerical methods for problem solving. We have constructed such a system: IRENA — an Interface between REduce and the NAg library [Dewar 1989, Dewar & Richardson 1990]. It has the following properties:

- It allows a Reduce user access to the full range of NAG routines;
- It provides an easy–to–use, interactive front–end to the NAG library;
- It provides a basic code–generation facility for users of the NAG library.

3.1 Simple use of IRENA

IRENA is used interactively within the normal Reduce environment, so that on first inspection it might appear similar to a set of native Reduce procedures, although this is not really the case. As our first example let us consider the NAG routine D01AJF which calculates an approximation to the integral of a function $f(x)$ over the interval (A, B). Despite the fact that it is one of the simplest routines in the whole library, it still requires the user to provide twelve parameters as follows:

- Two real[1] scalars `A` and `B`, the limits of integration;

[1] by *real* we mean either `REAL` or `DOUBLE PRECISION` objects.

```
2: d01ajf(a=0,b=1,f(x)=(log(x)-1)^-10);

{ALIST,BLIST,ELIST,RLIST,ABSERR,RESULT,DIVISIONS}

3: result;

0.098 92913 26406 1784

4: abserr;

2.003 05847 21634 49 E  -13
```

Figure 3-1: A simple example using IRENA to solve an integral.

- F, a user–defined function evaluating the integrand at a single point X;
- Two reals, EPSABS and EPSREL, which control the accuracy of the result;
- Two arrays IW and W which are used as working space, and their dimensions LIW and LW. On exit various elements of the arrays contain intermediate results and values, which may be of use if the computation has failed in some way;
- Two reals, RESULT and ABSERR, which on exit contain the approximation to the integral and an estimate of the absolute error respectively;
- IFAIL, an integer, the standard NAG diagnostic parameter.

Suppose that we want to solve the problem:

$$\int_0^1 \frac{dx}{(\log(x)-1)^{10}}$$

(an integral which Reduce is unable to solve symbolically since no elementary integral exists [Davenport *et al.* 1988]). The session with IRENA is shown in figure 3-1. Note that the argument to the IRENA function is a list of keys, rather than a straightforward list of parameters, and that only three of the twelve parameters required by the NAG subroutine have been provided by the user.

3.2 NAG Parameters.

Apart from those parameters used simply to return results, we consider there to be three distinct kinds of NAG parameter.

1. Those which are logically essential to the mathematical description of the problem. In this example the *data* parameters are `A`, `B` and `F`.

2. Those which *control* the operation and termination of the algorithm, in this case `EPSABS` and `EPSREL`.

3. Those which are used internally by the FORTRAN routine, such as workspace arrays and dimension arguments: in this case `W`, `IW`, `LW` and `LIW`. In fact anything which is not part of the statement of the problem or a constraint on its mode of solution we regard as such a *housekeeping* parameter. We also include `IFAIL` in this category (see section 1.1.1).

In principle the user must provide values for parameters in category 1, while defaults are provided for all the others. A default may be freely over-ridden simply by providing an appropriate value in the call to IRENA; a very real possibility for items in category 2, but an unlikely one for items in category 3, since these tend to be dependent on the "size" of the problem. The provision of appropriate defaults is discussed in chapter 4.

In practice parameters often fall into several categories. This happens in two ways:

- The amount of workspace allocated controls how many iterations of the algorithm are performed, i.e. it continues until it runs out of space. For instance, the parameter `LIW` in the previous example controls how many times the algorithm will subdivide the range (A, B) and so is really a control parameter, even though it is the dimension of a workspace array. There are relatively few such parameters in the Library.

- One array parameter may be used for several purposes. For example D02YAF has a parameter `W` whose columns are used variously for inputting data, returning results, and workspace. How we disentangle such a parameter is discussed in chapter 5.

3.3 Returning Results

The IRENA function returns a list of output parameters, whose values may then be inspected or manipulated as required. The parameters are called e.g. *d01ajf-result* rather than *result*, but may be referred to by the shorter name[2]. In this example the routine has returned a number of parameters which were not listed in our earlier exposition of the FORTRAN interface. These are, in fact, the "interesting" elements of the arrays W and IW. The mechanism by which they are transformed, and the rationale behind it, are given in chapter 5.

3.3.1 IFAIL

Finally, it is worth noticing that the user did not bother to check the value of the diagnostic parameter `IFAIL` on exit. A conventional user of the NAG library would regard this as an extremely bad (though one suspects rather widespread) practice. IRENA however does the checking itself and, if an error has occurred, notifies the user by displaying a warning together with any extra information output by the routine, along with the error diagnosis from the manual. For example, figure 3-2 shows what happens if IRENA is asked to integrate $1/x$ near the origin. As the results are sometimes useful even if IRENA has signalled an error, they are still returned in the usual way. In this case the problem is not tractable, but some of the returned values might help a user decide what remedial action to take. By examining ALIST and BLIST an experienced user could determine the location of the difficulty. The value of `IFAIL` is also returned. Some NAG routines make special provision for a restart after an error has occurred, a feature which IRENA can make use of by providing the extra keyword *reenter* in the appropriate call (see figure 10.2).

[2]This is to prevent values with common names being overwritten by subsequent calls to different routines.

```
5: d01ajf(a=0,b=1,f(x)=1/x);

 ** The maximum number of subdivisions (LIMIT) has been reached:
    LIMIT =          252   LW =         2000   LIW =          252
 ** ABNORMAL EXIT from NAG Library routine D01AJF: IFAIL =      1
 ** NAG soft failure - control returned

  The maximum number of subdivisions allowed with the given
  workspace has been reached without the accuracy requirements
  being achieved. Look at the integrand in order to determine the
  integration difficulties. If the position of a local difficulty
  within the interval can be determined (e.g. a singularity of the
  integrand or its derivative, a peak, a discontinuity...) one
  will probably gain from splitting up the interval at this point
  and calling the integrator on the subranges. If necessary,
  another integrator, which is designed for handling the type of
  difficulty involved, must be used. Alternatively consider
  relaxing the accuracy requirements specified by EPSABS and
  EPSREL, or increasing the amount of workspace.

{ALIST,BLIST,ELIST,RLIST,ABSERR,RESULT,DIVISIONS}
```

Figure 3-2: An example of how IRENA handles an error in the NAG routine.

3.4 Providing Values

So far we have shown how to provide values via the key list. In fact there are other ways to provide values for IRENA:

- Taking values from the global REDUCE environment. It is possible to put a key of the form $(\ldots, a = b, \ldots)$ in the parameter list, which will give the IRENA parameter a the value of REDUCE identifier b. Similarly a key of the form $(\ldots, a = a, \ldots)$ makes perfect sense, but it is syntactically cleaner to set the switch ENVSEARCH ON, which makes IRENA take any undefined values from the global REDUCE environment.
- Prompting for values. With the switch PROMPTVAL ON, IRENA will prompt for the values of any undefined parameters (which must be given in key form).

These two switches may be combined, in which case the order of priority is:

1. Values given in the key list.
2. Values given in the defaults file(s).
3. Values from the REDUCE environment.

before prompting takes place (see also § 5.5.1).

The reason for this hierarchy is to ensure that, if the user provides no parameters, he or she will be prompted for data values only. Hence values in the defaults files may only be over-ridden in the key list. An example of this alternate method of calling D01AJF is given in figure 3-3. This example also demonstrates the alternative names and forms of parameters described in chapter 5.

Values of parameters given in the key list are local to the call and do not interfere with, or affect, the values of REDUCE parameters with the same name.

3.5 Matrices

Reduce represents matrices by lists of list, with each value being explicitly stored, even if it is zero. It is possible to perform arithmetic on these matrices, and carry out various

```
3: d01ajf();

Please supply values for the following variables using the
usual key-line syntax. '@' may be used instead of the name
of the current variable.  Terminate each input with a ';'
character.  Replying ';<cr>' to any prompt will abort the
call.

(rectangle) REGION? @=[0:1];
(function) INTEGRAND? @(x)=4/(1+x^2);

{ALIST,BLIST,ELIST,RLIST,ABSERR,RESULT,INTERVALS}

4: result;

3.1415926535898
```

Figure 3-3: Letting IRENA prompt for all the data parameters.

operations such as taking their transpose or inverse. However, numerical computations often deal with very large arrays with special structures. Representing these densely as Reduce does is extremely wasteful of space, not to mention tedious to enter for the user. Moreover it is hard to write Reduce matrices to a file to be re-read later. Thus we have provided a special suite of functions to handle matrices in IRENA.

[Richardson 1988] has identified the various types of matrix which are useful to users of the NAG Library. We provide a function to input each type of matrix and store the information on the appropriate property list. The syntax is:

<matrix-type> <name> <elements>

An example of how to declare *tm* to be an upper-triangular matrix is given in figure 3-4.

These representations cannot be explicitly manipulated or displayed, so we provide a function to translate them to Reduce matrices as shown. We also provide functions to read and write IRENA matrices from and to files, and a function to convert Reduce matrices to IRENA ones. A complete list of IRENA matrices and their representations is in appendix C.

Ideally we would create a matrix domain [Bradford *et al.* 1986] to allow these

```
7: tri!-mat tm{{1,2,3},{4,5},{6}};

8: irena2reduce 'tm;

9: tm;

[1  2  3]
[       ]
[0  4  5]
[       ]
[0  0  6]
```

Figure 3-4: Declaring an upper-triangular matrix in IRENA.

matrices to be first class data objects. Unfortunately there are two problems with this:

- It is hard to create domains whose elements are themselves members of other domains, the only successful system known to the author being the TPS package [Barnes & Padget 1990].
- There can be multiple representations for the same matrix. For example, any matrix may be represented as a full matrix, or a sparse one. Deciding, for example, what representation to choose for the product of a full matrix and a sparse matrix requires some rigid definition of "sparseness". In this case a simple definition would be that a sparse matrix is one which can be represented more compactly in that form, i.e. where less than one third of the elements are non-zero. This does not, however, prevent a user from providing an inappropriate format. Checking the structure of a large matrix explicitly is time-consuming.

These problems are not insurmountable, but for the moment we see the matrix functions solely as a convenient method for inputting data.

3.6 Code Generation

IRENA works by generating a FORTRAN program to call the NAG Library (the mechanism is described fully in chapter 7). The code produced is as portable as possible and, through the use of the switch DOUBLE and the command PRECISION, code can be tailored for a target machine other than the current platform. In this case the user must also disable the compilation and linking phase described in chapter 7, by turning the switch CODEONLY on.

The ability to generate code to be used elsewhere is particularly useful when this code includes parameters which are themselves subprograms. Coding these in FORTRAN can be a particularly tedious and error-prone task, and their generation is discussed in chapter 6.

3.6.1 Machine Dependent Quantities.

For many implementations of numerical algorithms there are a number of parameters which are dependent on the underlying hardware. These include the machine precision, the largest positive floating point number and so on. The NAG Library contains a number of functions to return such values: for example the routine X02AJF returns the machine precision and X02ALF returns the largest positive floating point number. There are also routines which return an estimate of the active set size in a paged environment or indicate how the system treats underflow. Such functions are essential if a user is to write portable code, and since one use of IRENA is to generate code on one machine (e.g. a desk-top workstation) to be run on another (e.g. a mainframe) it is essential that the code it produces uses them. The way such quantities are treated in IRENA is as follows: at the Reduce level they are treated as constants so that, for example, the largest floating point number is called *fphuge*, but, at the code generation stage, we generate assignments for the returned function values of any of these quantities used, along with the necessary type declarations and `EXTERNAL` statements.

As well as the above machine constants there are various mathematical constants whose values depend on the precision of the implementation. With the exception of e, which is replaced by `EXP(1.0)` (or its double precision equivalent), IRENA handles

these in the same way. The most obvious example is *pi*.

A complete list of the objects IRENA treats in this way is given in appendix D.

3.6.2 Optimisation.

If the switch GENTRANOPT is turned on, IRENA will pass the expressions she generates through the symbolic optimiser. This is particularly useful when generating code for Jacobians, and even for one-shot problems can sometimes give some speed up.

3.7 Summary

In this chapter we have given a basic description of the capabilities and operation of the IRENA system. We have indicated how it may be used interactively to call the NAG Library, and also how it may be used to produce machine-independent source code. In later chapters we shall describe how the system actually works, together with the mechanism we have developed to customise and improve its interface.

Chapter 4

Defaults

We have seen how we categorise NAG parameters as being data, algorithmic control, or housekeeping; and that we provide default values and expressions for those in the latter two groups. This chapter describes in more detail the mechanism which we use.

4.1 Defaults Files

Default values for NAG parameters are provided in various defaults files. Each routine may have a system-provided default file, which contains the recommended default values and expressions and, optionally, a defaults file provided by an individual user. A user's own defaults will over-ride the system's.

Defaults may be values or expressions. There are also a set of error tolerances whose values can be changed at the top level. These are **userabserr**, **userrelerr**, **usermixer**, and **userinputerr**, which are used for absolute, relative, mixed and input errors[1] respectively. The standard starting value for each one is currently 10^{-5}, but this can be changed at any time[2]. These are the only quantities whose values are fairly invariant between routines — it makes no sense, for example, to specify a maximum number of iterations which applies to every routine in the library.

IRENA provides default values for all control and housekeeping parameters. Those in

[1] i.e. an estimate of the maximum error in some data values.

[2] Either interactively during a Reduce session, or (in the PSL implementation of Reduce) in the user's startup (*.reducerc*) file.

the first category — tolerances, iteration limits etc. — tend to be straightforward values, while those in the second — array bounds mainly — tend to be functions dependent in some way on the *size* of the data parameters. For example the amount of workspace required to solve a set of multivariate polynomial equations may depend on the number of such equations and the number of variables.

Most default values have been determined by careful reading of the Library documentation, or consultation with NAG experts. The preparation of the first generation of defaults files is being carried out by NAG and in future they intend to explicitly incorporate such "recommended values" in the main documentation for each routine. When new routines are incorporated into the library the authors will be asked to incorporate such details. Thus in future releases of IRENA the defaults files could be generated at least partially automatically.

Occasionally NAG routines incorporate a default parameter, indicated by setting it to a "silly" value. For example the quadrature routine D01AHF has a parameter `NLIMIT` which limits the number of function evaluations the algorithm may make. If `NLIMIT`≤ 0 then the default value of 10000 is used. In these cases we incorporate the latter value, to help in the generation of the IRENA documentation (see section 8.4.4). Some parameter values are heavily influenced by the fact that IRENA is an interactive system. The most obvious examples are the diagnostic print parameters, where we feel that potential users will not want to see screens of information flashing in front of them.

4.1.1 The Defaults Files' Syntax.

The syntax for the defaults files is given in figure 4-1. An explanation of the notation used may be found in appendix A. Each entry consists of the parameter followed by a colon and its default. It allows for conditionals, arithmetic, and the extraction of certain types of information from input parameters:

- We can extract the number of parameters a particular user-defined function (or family of functions) takes using the operator *PARAMS*.
- We can determine the number of functions in a family of functions through the operator *MULTIPLICITY*.

```
<default>          ::=  <comment> || <scalar-default> || <matrix-default>
<comment>          ::=  % <text>
<scalar-default>   ::=  <name> : <clause>
<matrix-default>   ::=  <name> ( <dim> ): <clause>
<dim>              ::=  * || IDENTITY || <expr0> { , <expr0> }
<clause>           ::=  <conditional> || <expr0>
<conditional>      ::=  if <bool-clause0> then <expr0> {else <expr0> }
<bool-clause0>     ::=  <bool-clause1> { OR <bool-clause1> } || <have> ||
                        REENTER || <matrixp> || <scalarp>
<bool-clause1>     ::=  <bool-clause2> { AND <bool-clause2> }
<bool-clause2>     ::=  <bool-clause3> { <relational> <bool-clause3>}
<bool-clause3>     ::=  { NOT } <expr0> || NOT( <bool-clause0> )
<relational>       ::=  = || ~ = || < || > || <= || >=
<expr0>            ::=  <expr1> { <addop> <expr1> }*
<addop>            ::=  + || -
<expr1>            ::=  {<addop>} <expr2> { <multop> <expr2> }*
<multop>           ::=  * || /
<expr2>            ::=  <expr3> { ^ <expr3> }*
<expr3>            ::=  ( <expr0> ) || <abs> || <dimension> || <string> ||
                        <minimum> || <maximum> || <length> || <params> ||
                        <multiplicity> || <nth-root> || <tolerance> || TRUE ||
                        FALSE || UNSET || <identifier> || '<identifier> ||
                        <irena-constant> || <irena-function> || <number> ||
                        <matrix-element> || CANCELDEFAULT
<have>             ::=  HAVE ( <name> )
<matrixp>          ::=  MATRIXP ( <name> )
<scalarp>          ::=  SCALARP ( <name> )
<dimension>        ::=  DIM ( <name> {, (1 || 2)} )
<length>           ::=  LENGTH ( { <name> || <matrix-element> } )
<minimum>          ::=  MIN ( { <vector> || <list-of-numbers> } )
<maximum>          ::=  MAX ( { <vector> || <list-of-numbers> } )
<list-of-numbers>  ::=  <expr0> { , <expr0>}*
<abs>              ::=  ABS ( <expr0> )
<params>           ::=  PARAMS ( <name> )
<multiplicity>     ::=  MULTIPLICITY ( <name> )
<nth-root>         ::=  NTH!-ROOT ( <expr0> , <expr0> )
<tolerance>        ::=  !*USERINPUTERR!* || !*USERABSERR!* ||
                        !*USERRELERR!* || !*USERMIXERR!*
<irena-function>   ::=  <name> ( <arguments> )
<arguments>        ::=  <list-of-numbers> || <vector>
<matrix-element>   ::=  <name> ( <expr0> { , <expr0> } )
```

Figure 4-1: Syntax for the IRENA defaults files.

- We can determine an array's dimensions (or its first or second dimension[3]) with the operator *DIM.*
- We can determine the length of a string using the operator *LENGTH.*
- We can determine the minimum or maximum value in a list, or a user-defined vector.
- We can apply a user-defined function to a user-defined vector or list of numbers, for example to determine a suitable starting value for a parameter.
- We can signal that a parameter is optional by giving it the default value *UNSET.*
- We can check whether a parameter has been given a value using the operator *HAVE.*
- We can check whether an item is a global reduce matrix or scalar using the operators *MATRIXP* and *SCALARP*[4].

Defaults may be provided for matrices in several ways. Individual elements can be given default values, default values can be set for the whole matrix, or it may be set to the identity matrix. A matrix may also be given the default *UNSET*, to make it optional.

4.2 The defaults mechanism.

As stated earlier, in addition to the system default files, the user may create his or her own. Default values or expressions in the user's files take precedence over those in the system files, but both files are processed at runtime so the user need not provide values for all the parameters.

The system default files are located in the directory given by the REDUCE variable **system-defaults-directory.* If the user has any defaults files then the name of the directory where they are situated should be assigned to the REDUCE variable **defaults-directory*, either during the IRENA session or in the user's *.reducerc* file.

[3]Objects with more than two dimensions are usually specified using jazz functions (see § 5.5.1).

[4]This is useful when we are picking up structures output by a previous call to IRENA for input to a subsequent call.

4.2.1 Evaluating default expressions.

The order in which defaults are given in the defaults file is immaterial, so it is perfectly possible that a given expression cannot be evaluated at the moment it is processed. For example, if a defaults file contains the sequence:

```
M : N + 1
N : PARAMS(F)
```

then the value of `M` is initially undetermined (assuming that no value for `N` has been provided earlier). When this occurs, `M` is given the temporary value *STACKED*, and placed on the **unresolved-list*. Each entry on this list is of the form:

(<variable> <list of missing variables> <expression>)

in this case:

(M (N) (N !+ 1))

A second list, of the **required-values*, is also maintained. This associates the missing variables with the names of the unevaluated expressions which require them. Every time a value is determined, the lists are updated:

- If the parameter has an entry in **required-values* then that entry is removed, and each element associated with it has its list of missing variables updated.
- As soon as a list of missing variables is empty, that expression is evaluated and the process repeated.

There is a slight modification in the case of a conditional. If we cannot evaluate the boolean switch, then we stack the whole clause until we can, without bothering about whether we have all the necessary information for either branch. Thus we may have to restack a branch which we cannot evaluate after we have evaluated the switch. However this does avoid waiting needlessly for a value which we do not actually need.

Values for expressions which cannot be evaluated are determined in the usual way, depending on the settings of *PROMPTVAL* and *ENVSEARCH*. Since the defaults

have priority here it allows us to prompt the user for only the non-defaulted (i.e.*data*) parameters. As these values are determined the **unresolved-list* is checked as before.

It is possible for a deadlock situation to arise if the defaults files have been badly written. In practice this will normally be due to an inappropriate entry in the user's default file. Suppose, for example, that the system default file contains the expressions:

```
M : N + 1
N : PARAMS(F)
```

while the user's file contains the statement:

```
N : M - 1
```

with no default for `M`. Then the default value for `N` in the system defaults file will be ignored, and IRENA will be unable to determine values for either `M` or `N`. If *PROMPTVAL* is on, the system will resolve this conflict by prompting for either `M` or `N` (the order is arbitrary) and then determining the other value automatically. If *PROMPTVAL* is off, the call will terminate with an error.

4.2.2 Cancelling System Defaults.

It is possible that a user might wish to ignore permanently the system defaults, e.g. to ensure that a value is prompted for if it is not provided in the keyline. This can be done by placing a command of the form:

```
FOO : CANCELDEFAULT
```

in the user's personal default file. This might be useful in the case where a set of routines were being customised for a specific group of users, e.g. in the teaching of numerical analysis.

4.3 Example.

The defaults file for the routine D02RAF is given in figure 4-2. D02RAF solves the two-point boundary value problem with general boundary conditions for a system

```
      N : multiplicity(FCN)

      MNP : 128

      NP : 17

      NUMBEG : multiplicity(GBEG)

      NUMMIX : multiplicity(GMIX)

      TOL : !*userabserr!*

      INIT : if Y = unset then 0 else 1

      X(*) : if INIT = 0 then unset

% A and B are scalars defined in the jazz file

      X(1) : A

      X(NP) : B

      Y : unset

      IY : max(dim(Y),N)

      IJAC : 1

      DELEPS : 0

      LWORK : MNP*(3*N*N + 6*N + 2) + 4*N*N +3*N

      LIWORK : if IJAC = 0 then MNP*(2*N + 1) + N*N + 4*N + 2
                           else MNP*(2*N + 1) + N

% IFAILB and IFAILC are scalars defined in the jazz file

      IFAIL : 100*IFAILC + 10*IFAILB + 1

      IFAILB : 1

      IFAILC : 0
end;
```

Figure 4-2: The defaults file for D02RAF.

of ordinary differential equations in the interval (a, b). The FORTRAN interface has twenty-four parameters, including six user-supplied subroutines. The IRENA interface will be described in § 5.7, after the concept of *jazzing* has been introduced in chapter 5. We shall give a description of the entries in the defaults file here, however.

`N` The FORTRAN parameter `FCN` is a subroutine which evaluates the family of differential equations at a general point. In IRENA this is represented by a set of functions. `N` is the number of such functions, denoted by the *multiplicity* of `FCN`.

`MNP` This parameter represents the maximum permitted number of points in the finite-difference mesh, and cannot be less than 32. The value 128 given in the defaults file was recommended by an expert.

`NP` This parameter represents the number of points in the initial mesh and must lie between 4 and `MNP`. The value 17 was recommended by an expert.

`NUMBEG` The number of left-hand boundary conditions. These are provided as a series of functions called *gbeg*.

`NUMMIX` The number of coupled boundary conditions. These are provided as a series of functions called *gmix*.

`TOL` An error tolerance, which is given the value **userabserr** described in § 4.1.

`INIT` The user may optionally provide an initial mesh and approximate solution as the arrays `X` and `Y` respectively, in which case `INIT` must be given a non-zero value. The clause in the defaults file checks this by looking to see whether `Y` has its default value (*unset*) or not.

`X` If `INIT` is non-zero then `X` must contain an initial mesh, otherwise the extreme values must be set to the lower and upper endpoints of the interval (a, b) respectively. The effect of the first statement in the defaults file for `X` is to say that if `INIT` is 0 then elements of `X` need not be given values. The other two statements set the extreme values of `X` to `A` and `B` which are actually new objects created by IRENA.

Y This parameter need not always be provided as explained above.

IY This is the dimension of Y, which must be at least N to allow a solution to be returned. If INIT is non-zero then the actual size of Y, if larger, will be taken.

IJAC This is a flag which denotes whether or not the user has provided Jacobian evaluation routines. As IRENA always generates these the default value is 1.

DELEPS This specifies whether continuation is required. By default it is not.

LWORK The length of the real workspace array, dependent on the number of equations and the size of the mesh.

LIWORK The length of the integer workspace array which varies depending on whether the user has supplied Jacobian evaluation routines or not.

IFAIL, IFAILB, IFAILC D02RAF is one of the few NAG routines which does not use the standard system for the IFAIL error flag. In this routine, IFAIL should be a three-digit integer where each digit can be either zero or non-zero to specify the form of failure (hard or soft), whether error messages should be printed or not, and whether warning messages should be printed or not. IRENA splits the latter two off as separate parameters, and by default enables error messages but disables warnings.

Chapter 5

The Jazz System

As previously stated, the form of input parameters is changed by the *jazz* system. In this chapter we describe the various cases which can occur, and how the process of *jazzing*: i.e. the mapping of objects between their NAG and IRENA representations, is handled. Each routine has a jazz file associated with it, which contains instructions to IRENA on how to interpret non-NAG parameters it may meet, and how to get values for NAG parameters from their jazzed components. There are two classes of jazzing: *input* and *output*, which act on input and output parameters respectively. In the first case the jazzed form of parameters is not compulsory and there may be multiple jazzings for any parameter, so that a user who is familiar with the Fortran routine may still use the old parameter names and definitions; whilst particular interfaces may be specially defined for specific sets of users. Output jazzing is compulsory, since not only does it make no sense to return the same item in several different ways[1], but to do so would often entail creating extremely large structures, most of which would be redundant. A useful side effect of the jazz system is that, in cases where the same FORTRAN parameter is used for both input and output, in IRENA they are distinct. The user has the option of making data parameters either global or local in scope, giving more flexibility.

There is one other area where the IRENA interface differs markedly from that of the NAG Library, and that is in how parameters which are either Functions or

[1] though it is possible to force this should the need arise.

Subroutines are defined. We differentiate between this process and the jazz system for purely functional reasons: conceptually they are identical. The mechanism for handling such parameters is described in Chapter 6.

None of these processes affect the Fortran code which IRENA generates, as the job of transforming the parameters is done at the Lisp level. This fact is important if the code is going to be reused as part of another application.

5.1 Input Jazzing

5.1.1 Aliases

The simplest way in which jazzing is used is to provide different names for parameters. There are a number of cases where this is useful:

- Fortran restricts names to six characters, and therefore these are often not very meaningful;
- It is preferable to have the same names for equivalent parameters across a whole chapter, and indeed in some cases across the whole Library;
- Different groups of users use different terminology.

There can be several aliases for the same object, or even aliases to aliases, and of course the user is still free to use the original parameter name, if desired. In addition to the system provided aliases, we allow the user to set up his or her own alias file (see § 5.5.3).

5.1.2 New Scalars

Sometimes the NAG parameter is not the natural parameter. For example the NAG routine E02ADF, which computes least-squares polynomial approximations to an arbitrary set of data points, has a parameter KPLUS1 whose value is one plus the maximum degree required. This form of jazzing transforms the IRENA parameter, in this case k, to its NAG equivalent.[2]

[2]This contrasts with the situation where the NAG user might be expected to provide values for both *K* and *KPLUS1*. In this case we would class *KPLUS1* as a housekeeping parameter and give it

5.1.3 Keywords

Some NAG parameters can only take a limited number of values: for example `.TRUE.` or `.FALSE.`. In this case we define a set of keywords, each of which is equivalent to one of these cases. For instance, the routine E02BCF evaluates a cubic spline and its first three derivatives from its B-spline representation. It has a parameter, `LEFT`, which specifies whether left or right handed values are to be computed, depending on whether its value is 1 or not. IRENA has a pair of keywords *left* and *right*, which can be interpreted as `LEFT=1` and `LEFT=0` respectively. Thus a typical call to E02BCF would look like:

```
3: e02bcf(vec k {0,0,0,0,1,3,3,3,4,4,6,6,6,6},
3:        vec c {10,12,13,15,22,26,24,18,14,12,0,0,0,0}, x=0, right);
```

Normally we also provide the NAG parameter with a default value, so that one of the keywords is supplied for symmetry only.

Occasionally we use a keyword to force a different sequence of calls to the NAG Library, as in the example described in § 10.3.

5.1.4 Rectangles

NAG normally represents a rectangular region either as two scalars (in the one-dimensional case), or two arrays of lower and upper bounds[3]. In IRENA we define a rectangle to be a single object in its own right, consisting of a set of pairs of numbers surrounded by square brackets. For example the constraints on the variables in an optimisation routine can be given as a rectangle:

```
1: e04jaf(bounds = [ 1:3, -2:0, *:*, 1:3],
1:     vec start {3,-1,0,1},
1:     f(x1,x2,x3,x4)=(x1 + 10*x2)^2 + 5*(x3 - x4)^2
1:                                   + (x2 - 2*x3)^4 + 10*(x1 - x4)^4);
```

the default value $k+1$. This happens frequently in older routines where one parameter is an array dimension since, in FORTRAN-IV, array dimensions could not be expressions.

[3] Occasionally some of the bounds are functions, and are thus represented by subprograms. There are cases where the two representations are mixed, as in the multi-dimensional integration routine D01DAF.

Here *bounds* represents the constraints on $x1, x2, x3, x4$, i.e.

$$\begin{aligned} 1 &\leq x1 \leq 3 \\ -2 &\leq x2 \leq 0 \\ 1 &\leq x4 \leq 3 \end{aligned}$$

The asterisks indicate that $x3$ is unconstrained in both the positive and negative directions, as explained in § 5.1.5.

5.1.5 Very Local Constants

Sometimes NAG attaches special meanings to certain values. For example, in the example shown in § 5.1.4, the FORTRAN arrays `BL` and `BU` contain the lower and upper constraints on the values of the x_i. If the value given is a very large negative or positive number respectively, then this is taken to mean that the value of that particular x_i is unconstrained in that particular direction. In the example x_3 is completely unconstrained, and in the rectangle *bounds* its constraints are denoted by asterisks. Each asterisk in fact means something different. For the upper bound it means *fphuge* — the largest floating point number (see § 3.6.1) — while for the lower bound it means *-fphuge*. The asterisk is a *very local constant*, and it enables us to provide a uniform interface within a routine. In general the asterisk character is interpreted as meaning that a parameter is "unset", i.e. not given a value.

5.1.6 Jazzing Matrices

There are three main reasons for jazzing arrays on input:

1. NAG arrays are sometimes confusing, with different columns being used for different purposes (e.g. `W` in D02YAF which has a variable number of columns used for inputting values of derivatives, returning results, and workspace). Jazz allows the user to specify their separate logical components and then assembles them correctly.

2. Matrices with a special structure are represented by NAG routines in a multitude of ways to make efficient use of memory, for example two triangular matrices might be packed into one FORTRAN array to save space. However, we have provided representations for matrices which preserve these structures, and so need to transform the IRENA or Reduce representation to the NAG one. Note that we do not insist that e.g. a triangular matrix be represented explicitly as an IRENA triangular matrix, jazz will try to coerce any IRENA or Reduce matrix to the required type.

3. NAG routines often expect the user to provide a large array, only some of whose elements are set. This is usually to allow the routine to manipulate elements of the array in place, rather than using separate workspace. An example is `MU` in E02DAF whose first and last four elements are zeros. It is nicer to allow the user to provide the smaller structure, which jazz then "pads out" to the larger one.

5.1.7 Complex Objects

The FORTRAN standard [ANSI 1978] is somewhat deficient in its representation of complex numbers. Whilst it provides a single precision complex data type, there is no double precision analogue. This has led to a great deal of inconsistency between the available compilers: some follow the standard and offer no double precision complex data type at all; while others do, but call them by a variety of names (`double complex` on SUNs, `COMPLEX*16` on IBM machines etc.). As a result, the implementors of algorithms have chosen a variety of representations for complex objects. For scalars the normal ones are:

- a pair of scalars representing the real and imaginary parts;
- a vector of length two, containing the real and imaginary parts (and corresponding to the normal implementation of the complex data type);

while for arrays the common representations are:

- a pair of arrays representing the real and imaginary parts;

- a single vector, those elements with odd indices being the real parts and those with even ones being the imaginary parts (again, this corresponds to the normal implementation of the complex data type);

There are a few rather more obscure representations, mainly where we are dealing with an array with some special structure (e.g. a factored Hermitian matrix).

In IRENA we expect the user to provide a normal Reduce complex object, which we will then convert to the appropriate format.

5.1.8 Unpacking Matrices

Most jazzing so far has consisted of taking small logical objects and constructing larger FORTRAN objects from them. Occasionally however, we would like to take a matrix provided by the user and make each element into a FORTRAN scalar. For example, the NAG routine C02AJF finds the roots of a quadratic equation with real coefficients using the well known formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where the coefficients a, b, c are provided by the user as three scalars. However, for consistency with the rest of the C02 chapter, we would like the user to be able to provide them as a vector called *coefficients*. Hence the interface can be either:

```
1: c02ajf(vec coefficients{10,3,1});
```

or:

```
2: c02ajf(a=10,b=3,c=1);
```

5.2 Output Jazzing

5.2.1 Matrices

There are three main uses for output jazzing of arrays:

1. Disentangling Fortran arrays into their logical elements. In § 5.1.6 we described W in D02YAF which has some columns used only for input and some for output. Clearly we want to return the output columns as separate structures.

2. Unpacking arrays to restore some structure to them. For example two triangular matrices are sometimes returned as a single array, IRENA unpacks them and returns the two matrices separately.

3. Trimming large structures into smaller ones. Analogous to the third case in § 5.1.6 we now wish to get rid of the excess padding we added earlier. Alternatively, we might wish to return just the "interesting" bits of a workspace array, as is the case with W in D01AJF, as described in chapter 3.

5.2.2 Complex Objects

On input we generally create real objects from given complex objects, on output we do the reverse and generate complex objects from the real data returned by the FORTRAN routine.

5.2.3 Packing objects into larger structures

Analogous to § 5.1.8, here we take several output parameters and turn them into an array. For instance in the example given the FORTRAN returns two vectors ZSM and ZLG, representing the two (complex) roots. We transform them into the array *roots*, for consistency with the rest of C02.

5.2.4 Output Aliasing

Analogous to the input case, we often wish to give FORTRAN output parameters more meaningful names. In some cases the actual name on output depends on the initial parameters chosen by the user or some output value[4]. Additional to the system provided aliases, we allow the user to set up his or her own alias file (see § 5.5.3).

[4]For example, depending on the input value of the parameter JOB, the routine C06EKF will return either a convolution or correlation in the array X.

5.3 Presentation of Results

5.3.1 Returning Input Parameters with the Output

Many routines require the user to provide an estimate of the accuracy to which the result is to be calculated. Some return an estimate of the final accuracy obtained, others will terminate with an error flag (normally a non-zero `IFAIL`) if this accuracy is not achieved. Since input tolerances are *control* parameters (see § 3.2) their values are normally provided by default, with the result that in some cases where no final accuracy estimate is returned by the routine the user may be unaware of the validity of the results. In these cases it is helpful to return the input tolerance as an output parameter.

5.3.2 Ordering the Output Parameters

Sometimes the "natural" order in which IRENA returns parameters does not reflect their relative importance. For example, in the D01JAF examples in chapter 3, the list of results has the relatively unimportant data from the workspace array `W` first, followed by the result and error estimate. There is, therefore, a facility to allow an interface designer to customise the order in which the output parameters are returned to the user.

5.4 The Ideal Interface

Not only are we trying to produce more natural interfaces to individual routines, but we are trying to produce consistent interfaces across whole chapters. For example to find the roots of a polynomial the user must provide its coefficients as a vector or matrix called *coefficients*, whichever routine is being used. Thus we have been forced, in some sense, to devise an ideal, canonical description of each problem domain. This has implications both for future implementations of algorithms, and also for other interfaces to the library. We will expand on this observation later.

5.5 The Jazz Mechanism

As stated earlier, each routine may have a jazz file associated with it. This contains instructions to the jazzing system on how to create the FORTRAN objects or return IRENA ones.

5.5.1 Input Jazzing

On input, scalar and matrix objects are handled differently. In the former case the actual NAG object is created: suppose for example that the jazz file contains the line:

```
{NEWSCALAR} kplus1 [k+1] : k
```

This tells IRENA that k is a new parameter introduced as a means of providing the value of the FORTRAN parameter `KPLUS1`. So, if the user gives k a value in the key list, the parameter *kplus1* will be given the value $k + 1$. No record will be kept of the value of k.

The problem with doing this for matrices is that we may end up generating very large structures which are never going to be seen by the user, but will just be translated into FORTRAN and then thrown away. Another problem that occurs with matrices is that we encounter a multitude of slightly different representations in different routines. Thus we need an easily extensible, modular mechanism to handle them.

Each type of matrix jazzing is known as a *jazz function*, and has three functions associated with it:

check-function A function to check that the user has provided all the necessary components of the FORTRAN parameter. This function may also be used to find out which components are still missing (e.g. for prompting).

dim-function A function to return the array dimensions of the FORTRAN object, for use by the defaults system (see Chapter 4).

trans-function A function to generate the assignment statements for the FORTRAN array, using a suite of specially-provided commands.

Reconciling Conflicts

Jazzing is not compulsory and, since objects may be provided globally as well as in the key list, it is possible that we may have several alternative representations for the same object. In this case we decide which is the correct one as follows:

- If the object has complex jazzing, i.e. the FORTRAN object is the real or imaginary part of an IRENA one, and the complex object was provided in the key list we use that.
- If the FORTRAN object was provided in the keyline then we use that.
- If the object has a jazz function and all its components are present either globally or in the key list then we use that.
- If the object has complex jazzing and the complex object was provided globally then we use that.
- If the FORTRAN object was provided globally then we use that.

Note that providing the value of an object multiply in the key list will cause an error (e.g. providing both *k* and *kplus1* in the example in § 5.5.1).

5.5.2 Output Jazzing

Most of this is signalled by straightforward commands in the jazz file. For example the line:

```
{OUTPUT} W#1 : result
```

means return the first column of the FORTRAN array `W` as the Reduce matrix *result*. The various options for getting at parts of an array are:

- Return one element.
- Return one column.
- Return bits of a vector as a vector.

- Return a rectangular portion of an array (i.e. trim the edges).

Any number of IRENA output parameters may be constructed from one FORTRAN object, and indeed the same part of a FORTRAN object may be part of several IRENA ones.

There are also several types of unpacking which can be signalled, to restore structure to objects. For example getting two symmetric matrices from one array. However it became clear that there were many variations on these, and so an extensible, modular mechanism has been adopted to handle this case as well. These *out functions* take a FORTRAN object and create any number of IRENA objects with the aid of a suite of special functions. They return a list of all the objects they have created.

Dependencies

Sometimes it is necessary for output jazzing to be done in a specific order. For example we have seen that D01AJF (like many other quadrature routines) returns matrices of intermediate results as elements of the real workspace array `W`. These can be very useful if the computation fails, since by inspecting them it is often possible to determine the location of a singularity or discontinuity. The lengths of these matrices depend on the number of times which the routine has subdivided the range of integration, which is returned as the first element of the integer workspace array `IW`. Hence we would like to process `IW` before we process `W`. This can be ensured through the use of a *PRECEDENCE* statement, so that part of the jazz file for D01AJF looks like:

```
{precedence} IW
{output} IW(1) : divisions
{output} W[1 : divisions] : alist
{output} W[divisions+1 : 2*divisions] : blist
{output} W[2*divisions+1 : 3*divisions] : elist
{output} W[3*divisions+1 : 4*divisions] : rlist
```

```
<user-alias>   ::=  <in> || <out>
<in>           ::=  {IN}  <identifier> : <identifier>
<out>          ::=  {OUT}  <identifier> : <identifier>
```

Figure 5-1: Formal syntax for the users alias files.

```
% Alias file for D01AJF
{IN} a : lower
{IN} b : upper
{OUT} result : quad
{OUT} abserr : error
end;
```

Figure 5-2: An example user alias file.

The *PRECEDENCE* command can take a list of objects instead of just one, and there may be multiple *PRECEDENCE* statements in a single jazz file. Those objects in a list have equal precedence, and greater precedence than any in statements further down the file. Thus if necessary the order of output jazzing can be precisely dictated.

This mechanism is more efficient than a stacking procedure such as that used with the defaults files, and is preferred since jazzing is not under user control.

5.5.3 User Jazzing

Because of the complexity involved in jazzing, we do not allow the user access to the full system. (However an informed user could substitute their own set of jazz files for the system provided ones by simply resetting the value of the global Reduce variable **jazz-directory* which represents their location.) It is not unreasonable, however, to suppose that some users might want to change the names of some parameters, and so for this purpose we allow the user to set up alias files for each routine, should he or she wish.

The formal syntax is given in figure 5-1, and an example is shown in figure 5-2. A description of the notation used to describe the syntax can be found in appendix A.

5.6 Formal Jazz Syntax

The syntax for the jazz files is given in figure 5-3. Note that <expr0> refers to syntax given for the defaults files in figure 4-1. A description of the notation used can be found in appendix A.

<jazz-entry>	::=	<comment> || <precedence> || <scalar> || <vector> || <local> || <keyword> || <rectangle> || <newscalar> || <output> || <packed-array> || <jazz-fn> || <out-fn> || <template> || <complex-in> || <complex-out> || <append> || <unpack> || <cunpack> || <out-dims> || <alias> || <i2o> || <out-order>
<comment>	::=	**%** <text>
<precedence>	::=	**{PRECEDENCE}** <nag-name> {, <nag-name>}*
<out-order>	::=	**{OUTPUT-ORDER}** <nag-name> {, <nag-name>}*
<scalar>	::=	**{SCALAR}** <name> {, <name>}*
<vector>	::=	**{VECTOR}** <name> {, <name>}*
<keyword>	::=	**{KEYWORD}** <nag-name> **[** <expr0> {, <expr0>}+*n*] **:** <irena-name> {, <irena-name>}+*n*
<local>	::=	**{LOCAL}** <nag-name> **[** <expr0> **]:** <irena-name>
<rectangle>	::=	**{RECTANGLE}** <nag-name>,<nag-name>**:** { <irena-name> || <matrix-element> }
<newscalar>	::=	**{NEWSCALAR}** <nag-name> **[** <expr0> **]:** <irena-name>
<output>	::=	**{OUTPUT}** {<element> || <column> || <range> || <portion> } **:** <output-name>
<output-name>	::=	<irena-name> || <case-clause>
<case-clause>	::=	**CASE** <value> **(** <number> { , <number> }+*n* **)** <irena-name> { , <irena-name> }+*(n+1)*
<element>	::=	<nag-name>**(**<expr0>**)**
<column>	::=	<nag-name>**#**<expr0>
<range>	::=	<nag-name> **[**<expr0>**:**<expr0>{,<expr0>**:**<expr0>}*]
<portion>	::=	<nag-name>**[(**<expr0>,<expr0>**):(**<expr0>,<expr0>**)]**
<packed-array>	::=	<triangle> || <vect>
<triangle>	::=	<ttype> <nag-name> **:** <irena-name>
<ttype>	::=	**{ UPPER || LOWER || SUPPER || SLOWER || DIAG }**

<vect>	::=	<vtype> <nag-name> [<expr0> : <expr0>]: <irena-name>
<vtype>	::=	**{ LTRI || UTRI }**
<jazz-fn>	::=	{ <jazz-function> } <nag-name> : <arguments>
<out-fn>	::=	{ <out-function> } <nag-name> {, <nag-name>}* : <irena-name> {, <irena-name>}*
<template>	::=	**{TEMPLATE}** <routine> : <key-list>
<key-list>	::=	<keyword> {, <keyword>}*
<keyword>	::=	{ ~ } <key>
<complex-in>	::=	**{COMPLEX!-IN}** <nag-name> {, <nag-name>} : <irena-name>
<complex-out>	::=	**{COMPLEX!-OUT}** <nag-name> {, <nag-name>} {<portion>} : <irena-name>
<append>	::=	**{APPEND}** <nag-name> {, <nag-name>}* : <irena-name>
<unpack>	::=	**{UNPACK}** <nag-name> {, <nag-name>}* : <irena-name>
<cunpack>	::=	**{COMPLEX!-UNPACK}** <nag-name> {, <nag-name>}* : <irena-name>
<out-dims>	::=	**{OUT!-DIMS}** <nag-name> :'(<lisp-exp> . <lisp-exp>)
<i2o>	::=	**{I2O}** <nag-name> : <irena-name>
<alias>	::=	<atype> <name> : <name>
<atype>	::=	**{ KEY!-ALIAS || PROMPT!-ALIAS || SILENT!-ALIAS }**

Figure 5-3: Syntax for the IRENA jazz files.

5.7 Example Jazz File

The jazz file for D02RAF is given in figure 5-4. This routine is described in § 4.3 along with its defaults file. We shall first show how the standard NAG example from the manual is programmed in IRENA, and then describe in more detail how this is achieved by the jazz file. Suppose then that we wish to solve the differential equation:

$$y''' = -yy'' - 2\epsilon(1 - y'^2)$$

with boundary conditions:

$$y(0) = y'(0) = 0, \quad y'(10) = 1$$

```
% d02raf jazz file

{prompt!-alias} MNP : maximum_number_of_mesh_points

{key!-alias} MNP : mnmp

{prompt!-alias} np : initial_number_of_mesh_points

{key!-alias} NP : inmp

{output} NP : size_of_mesh_used

{prompt!-alias} NUMBEG : number_of_left_hand_boundary_conditions

{key!-alias} NUMBEG : nlhbc

{prompt!-alias} NUMMIX : number_of_coupled_boundary_conditions

{key!-alias} NUMMIX : ncbc

{scalar} a, b, ifailb, ifailc

{rectangle} a,b : range

{keyword} ifailb [0,1] : no_error_messages, error_messages

{keyword} ifailc [0,1] : no_monitoring, monitoring

{prompt!-alias} X : mesh

{output} X[1:out(size_of_mesh_used)] : mesh

{output} Y[(1,1):(N,out(size_of_mesh_used))] : solution

{output} ABT : absolute_error_estimates

{prompt!-alias} DELEPS : continuation_increment

{keyword} DELEPS [0] : no_continuation

{output!-order} size_of_mesh_used, mesh, solution,
                absolute_error_estimates, deleps

end; % of d02raf jazz file
```

Figure 5-4: The jazz file for D02RAF.

```
2: d02raf(range=[0:10],
2:        fcn1(x,y1,y2,y3,eps)= y2,
2:        fcn2(x,y1,y2,y3,eps)= y3,
2:        fcn3(x,y1,y2,y3,eps)= - y1*y3 - 2*(1 - y2*y2)*eps,
2:        gbeg1(y1,y2,y3,eps)= y1,
2:        gbeg2(y1,y2,y3,eps)= y2,
2:        gend1(y1,y2,y3,eps)= y2 - 1)$

{SIZE_OF_MESH_USED,MESH,SOLUTION,ABSOLUTE_ERROR_ESTIMATES,DELEPS}
```

Figure 5-5: An example of the use of D02RAF.

This can be rewritten in first order form to give the family:

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ y_3' &= -y_1 y_3 - 2\epsilon(1 - y_2^2) \end{aligned}$$

An IRENA program to solve this in the interval $(0, 10)$ using D02RAF is given in figure 5-5.

We can see that the three first order equations are given as *fcn1*, *fcn2* and *fcn3* while the left-hand conditions are called *gbeg1* and *gbeg2* and the right-hand condition *gend1*. Had there been any coupled conditions they would be called *gmix1* etc. The range of integration is called *range* and provided as a rectangle.

We shall now describe the function of the various lines in the jazz file. We start by setting up a number of aliases for objects. The *prompt-alias* command is used to set up a descriptive but verbose name to be used when the user is being prompted for a command. The *key-alias* is a shorter, more mnemonic name, to be provided by the user in the key list. The simple *output* clause for NP gives a different name (*size_of_mesh_used*) to be used to return that parameter.

We now declare some objects to be *scalar*, i.e. new objects used in the jazz and defaults files. The FORTRAN routine expects the range to be provided as the first

and last elements of the array X, however in IRENA we prefer it to be provided as a rectangle. We therefore declare a rectangle *range* whose elements will be called *a* and *b*. The FORTRAN routine also has an unusual way of setting IFAIL, described in § 4.3. We provide two new objects to reproduce the functionality of the composite IFAIL, and call them *ifailb* and *ifailc* (this naming scheme is a reflection of the way in which the NAG manual describes IFAIL). We then set up keyword names to represent the various legal values they can take. In the *ifailb* case, for example, providing the keyword *no_error_messages* will cause *ifailb* to be set to 0; while specifying *error_messages* gives it the value 1 (which is the default).

We also have some more complicated *output* statements. Only the portion of X which was used is returned; as it is a vector this is from the first element to that indexed by the output parameter *size_of_mesh_used*. This object is called *mesh*. The situation is similar with Y except that here we want a rectangular portion of a two-dimensional structure: from the first element of the first column to the element which is in row N (an input parameter) and column *size_of_mesh_used* (an output parameter). Specifying that we want the output value of a parameter in cases like this saves confusion when the parameter has different values on input and output.

Another keyword is set up for DELEPS, and finally the order in which the results of the routine should be returned is defined.

Chapter 6

Argument Subprograms (ASPs).

Argument subprograms are parameters to NAG routines which are themselves either functions or subroutines. They are used for a variety of purposes, as detailed below, and occur throughout the library. The nearest Reduce equivalent to a FORTRAN subprogram is an algebraic procedure, but to ask the user to provide ASPs in this form is unsatisfactory for the following reasons:

- The differences between the two languages make producing equivalent pieces of code difficult. FORTRAN uses call-by-reference semantics, while Reduce uses call-by-value. FORTRAN routines often return several results, while RLISP procedures can only return one (which may be a list). FORTRAN requires explicit type declarations, while RLISP only requires that variables be local (i.e. *scalar*) or globals (the default). A Reduce procedure can be translated into Fortran by Gentran, and the tokens *real* and *integer* can be used instead of *scalar* to give type information, but the two pieces of code will not be semantically equivalent. To generate correct code requires that the Reduce user understand FORTRAN, something we whole-heartedly wish to avoid.

- Matrices are treated in a rather clumsy fashion by Reduce: they can only be global variables, rather than local.

- We are trying to get away from the idea that the user needs to write a *program* to solve a problem, and to ask the user to encapsulate a mathematical object as

```
      DOUBLE PRECISION FUNCTION FUNCTN(NDIM,X)
      DOUBLE PRECISION X(NDIM)
      INTEGER NDIM
      FUNCTN=4*DEXP(DBLE(2*X(3)*X(1)))*(X(4)**2+2*X(4)*
     . X(2)+2*X(4)+X(2)**2+2*X(2)+1)**(-1)*X(3)**2*X(1)
      RETURN
      END
```

Figure 6-1: A simple ASP generated by IRENA.

a Reduce procedure would undermine that. We prefer to ask the user to provide these objects in their natural form: for example to generate an ASP which returns the value of a function at a given point, only the definition of the function is necessary.

Thus we provide alternative representations for all ASPs. In this way the ASP system is conceptually similar to the jazz system, the main difference being that in this case the alternative form is compulsory.

6.1 The User's View.

6.1.1 Function values.

The most common use of an ASP is to evaluate a given function or set of functions at an (arbitrary) point, e.g. an integrand or a set of differential equations. In IRENA the user is expected to supply a set of expressions corresponding to the functions, and the ASP system will generate the appropriate FORTRAN. For example a call to the integration routine D01GBF might look as follows:

```
1004: d01gbf( region=[0:1,0:1,0:1,0:1],
1004:         f(w,x,y,z)=4*w*y^2*e^(2*w*y)/(1 + x + z)^2);
```

The dummy parameters w, x, y, z are replaced by the correct FORTRAN parameters, in this case elements of the array `X`, during code generation to produce the code shown in figure 6-1.

```
2: d02bbf( range=[0:8],
2:         vec initialvalues {0.0,0.5,pi/5},
2:         derivative1(tt,y,v,phi)=tan(phi),
2:         derivative2(tt,y,v,phi)=-0.032*tan(phi)/v - 0.02*v/cos(phi),
2:         derivative3(tt,y,v,phi)=-0.032/v^2,
2:         vec output {1,2,3,4,5,6,7,8});
```

Figure 6-2: Using IRENA "subscript" notation for sets of functions.

```
      SUBROUTINE FCN(TT,Y,F)
      DOUBLE PRECISION TT,Y(3),F(3)
      F(1)=DTAN(Y(3))
      F(2)=-(0.02D0*Y(2)*DCOS(Y(3))**(-1))-(0.032D0*Y(2
     . )**(-1)*DTAN(Y(3)))
      F(3)=-(0.032D0*Y(2)**(-2))
      RETURN
      END
```

Figure 6-3: The FORTRAN produced from the "subscript" notation.

In many cases we are dealing not with a single function but with a set of functions, for example a set of differential equations. We provide two methods for entering such functions (note that these two methods may not be mixed for the same set of equations.) The first is analogous to the simple case above, while the second is useful for large sets of "related" functions. Mathematicians usually represent sets of functions with subscripts; in IRENA the equivalent notation is to suffix the function name with its index to produce a new name. This method is used in the key list as shown in figure 6-2 and produces the FORTRAN shown in 6-3.

An extension to more general functional notation is useful where we have a set of functions like:

$$f_i(x_1, x_2, \ldots, x_9) = -x_{i-1} + (3 - 2 * x_i) * x_i - 2 * x_{i+1} + 1$$

with appropriate modifications for the extreme values of i. In IRENA this is coded using *fsets*, either in the global Reduce environment, or in the key list. If set up in the global Reduce environment the values of the function may be inspected using the

fdisplay operator, as shown in figure 6-4.

With the following call to C05NBF (a routine to find the zero of a set of nonlinear functions):

```
5: c05nbf(vec x {-1,-1,-1,-1,-1,-1,-1,-1,-1});
```

the ASP `FCN` will be as shown in figure 6-5.

We use a loop to generate the values in case we are dealing with a really large set of assignments, which might overwhelm the compiler. Should the user elect to use the symbolic code optimiser however, explicit statements will be generated. An example of this is shown in figure 6-6.

FSETs may be indexed in more than one variable, for example:

```
1: fset g[1,j=1:4](x[1:4,1:4],y[1:4,1:4])=y(j)$
1: fset g[i=2:4,j=1:4](x[1:4,1:4],y[1:4,1:4])=x(i-1)*y(i)$
```

They may also refer to global Reduce parameters in their right hand sides. Care must be taken however, since for example referring to a general matrix element as follows:

```
8: m := mat((1,2,3,4,5,6,7,8,9,10))$
9: fset f[i=1:10](x[1:10])=x(i)*m(1,i)$
```

will cause an error at compile time, unless optimisation is turned on (since in this case each instance of $m(1,i)$ will be evaluated). The formal syntax for the *fset* operator is given in figure 6-7. A description of the notation used can be found in appendix A.

6.1.2 Jacobian and derivative values.

Quite often NAG routines require the user to write a routine to calculate the Jacobian or Hessian matrix, or the derivatives of a given set of functions. Although in theory an easy task, in practice errors are easy to make but difficult to detect. In IRENA we calculate derivatives automatically, expecting the user to provide only the original functions (which are normally required anyway for another ASP).

Because Jacobians and Hessians by their very nature consist of large numbers of related expressions, the efficiency of the generated code can be dramatically increased through the use of symbolic optimisation.

```
1: fset fcn[1](x[1:9])=(3-2*x(1))*x(1)-2*x(2)+1$

2: fset fcn[i=2:8](x[1:9])=-x(i-1)+(3-2*x(i))*x(i)-2*x(i+1)+1$

3: fset fcn[9](x[1:9])=-x(8)+(3-2*x(9))*x(9)+1$

4: fdisplay 'fcn;
                          2
FCN[1]= - 2*(X(2) + X(1)  - 3/2*X(1) - 1/2)

                          2
FCN[2]= - 2*(X(3) + X(2)  - 3/2*X(2) + 1/2*X(1) - 1/2)

                          2
FCN[3]= - 2*(X(4) + X(3)  - 3/2*X(3) + 1/2*X(2) - 1/2)

                          2
FCN[4]= - 2*(X(5) + X(4)  - 3/2*X(4) + 1/2*X(3) - 1/2)

                          2
FCN[5]= - 2*(X(6) + X(5)  - 3/2*X(5) + 1/2*X(4) - 1/2)

                          2
FCN[6]= - 2*(X(7) + X(6)  - 3/2*X(6) + 1/2*X(5) - 1/2)

                          2
FCN[7]= - 2*(X(8) + X(7)  - 3/2*X(7) + 1/2*X(6) - 1/2)

                          2
FCN[8]= - 2*(X(9) + X(8)  - 3/2*X(8) + 1/2*X(7) - 1/2)

FCN[9]= - 2*(X(9) + 1/2*X(8) - 2)
```

Figure 6-4: The use of *fset* and *fdisplay*.

```
      SUBROUTINE FCN(N,X,FVEC,IFLAG)
      DOUBLE PRECISION X(N),FVEC(N)
      INTEGER N,IFLAG,I
      FVEC(1)=-(2*X(2))-(2*X(1)**2)+3*X(1)+1
      DO 25001 I=2,8
          FVEC(I)=-X(-1+I)-(2*X(1+I))-(2*X(I)**2)+3*X(I
     .     )+1
25001 CONTINUE
      FVEC(9)=-(2*X(9))-X(8)+4
      RETURN
      END
```

Figure 6-5: Some FORTRAN produced from an *fset*.

```
      SUBROUTINE FCN(N,X,FVEC,IFLAG)
      INTEGER N,IFLAG
      DOUBLE PRECISION X(N),FVEC(N)
      T0=X(2)
      T1=X(1)
      T3=X(3)
      T7=X(4)
      T11=X(5)
      T15=X(6)
      T19=X(7)
      T23=X(8)
      T27=X(9)
      FVEC(1)=1+3*T1-(2*T0)-(2*T1**2)
      FVEC(2)=1-T1+3*T0-(2*T3)-(2*T0**2)
      FVEC(3)=1-T0+3*T3-(2*T7)-(2*T3**2)
      FVEC(4)=1-T3+3*T7-(2*T11)-(2*T7**2)
      FVEC(5)=1-T7+3*T11-(2*T15)-(2*T11**2)
      FVEC(6)=1-T11+3*T15-(2*T19)-(2*T15**2)
      FVEC(7)=1-T15+3*T19-(2*T23)-(2*T19**2)
      T33=-(2*T27)
      FVEC(8)=1-T19+3*T23+T33-(2*T23**2)
      FVEC(9)=4-T23+T33
      RETURN
      END
```

Figure 6-6: Some optimised FORTRAN produced from an *fset*.

```
<fset>          ::=  FSET <name> [<subscripts>](<parameters>) =
                     <expression>
<subscripts>    ::=  <subscript1> [, <subscript1> ]
<subscript1>    ::=  <integer> || {<identifier> = <integer> : <integer>}
<parameters>    ::=  [ <parameter1> { , <parameter1> }* ]
<parameter1>    ::=  <identifier> || { <identifier> [ <integer> : <integer> ] }
```

Figure 6-7: The Formal syntax for the *fset* operator.

6.1.3 Dummy Routines.

Sometimes NAG offers the user the choice of either writing their own routine, or of calling one in the Library. This is often the case when the routine is required to monitor the progress of the computation and output diagnostic information. In these cases (where appropriate) we automatically generate a dummy routine to call the NAG routine without further input from the user.

6.1.4 Output Routines.

Occasionally NAG routines require the user to provide a routine to output intermediate information during the execution of the algorithm. IRENA provides a procedure which may or may not require some input from the user, such as an array of points at which to generate diagnostics. Moreover the resulting information is available as an actual structure within Reduce, rather than simply printed out.

6.1.5 Matrix Manipulation Routines.

These are routines required to manipulate matrices in some way, often a specific (user-supplied) matrix for a given problem. In such cases IRENA requires that the user supply the relevant matrix, and the routine is generated using either Reduce's symbolic manipulation facilities, or a call to an appropriate NAG routine.

<requirements> ::= <functions> | <matrices> | <rectangle> | **NIL**
<functions> ::= **FUNCTION** [<function-list>]
<function-list> ::= { <name> | (<number>) | (<name> <number>) }*
<matrices> ::= { **MATRIX** | **VECTOR** } [<name>]
<rectangle> ::= **RECTANGLE** <number>
<number> ::= <integer> | <irena-variable>

N.B. Brackets of the form (...) denote a LISP list.

Figure 6-8: The syntax for an ASP's requirements.

6.1.6 Regions.

Some NAG routines are concerned with evaluating the endpoints of regions. We expect the user to express the region as an IRENA *rectangle* (see § 5.1.4), whose end points may be either expressions or constants.

6.2 The ASP system.

In the last section we described the different sorts of ASPS visible to the user. However for each apparent kind their are in fact many similar but distinct examples (more than seventy separate *types* in total). In this section we will describe the mechanism used to tell IRENA how to generate the FORTRAN for a specific ASP from the information given by the user. With each type of ASP we associate three things:

1. A set of requirements;
2. A lisp function to perform the code generation;
3. A Gentran template.

6.2.1 The Requirements.

The requirements of ASP type n is a list on the property list of the identifier *aspn*. The syntax for this list is as given in figure 6-8.

Where no name is given the default is the name of the particular subprogram being generated (which may of course be aliased to something else through the jazz system).

The requirements may be used to check that all the necessary objects have been provided before code generation commences in earnest. Notice that care must be taken here, since for given requirements e.g. *'(function (f n))* the value of n may well be calculated by seeing how many functions f_i exist. In this case, if PROMPTVAL is on, IRENA will prompt the user for n before prompting for the f_i. If PROMPTVAL is off then IRENA will stop and report an error[1].

6.2.2 The Templates.

ASPs are generated using GENTRAN templates which call a host of special functions. There are two types of templates: those designed to be used with a general class of ASPS, for example those returning the values of functions, and those which are used with individual ASPS which do not fit neatly into any particular class. The selection of the template is done by the ASP function, as described below. Values are passed to the template via global variables.

6.2.3 The ASP Functions.

The ASP functions do three things:

1. They declare the types and argument list of the subprogram.
2. They assign the values of relevant global variables (and clear any others that the template might look at).
3. They initiate the processing of the correct template.

The name of the subprogram being generated is assigned to the global variable **fun* by the procedure *getgen* which calls the ASP function.

[1]IRENA does not currently accept functions defined in the global Reduce environment.

Declaring Types.

This is done through two procedures. The first — *declare-asp* — declares the sort of subprogram, its type, and its list of parameters in the correct order. For example:

```
declare-asp(!*fun,'real,'function,'(x y));
declare-asp(!*fun,nil,'subroutine,'(ndim z nfun f));
```

The second procedure is the standard GENTRAN *declare* which is used to set up the parameter types.

Global Parameters.

The different types of standard template each require different global variables to be set. If these variables require no value for a particular call they should be set to NIL to avoid picking up previous values. The relevant variables for particular templates are given below. The values need not be constants, but may be the values of NAG parameters (which are guaranteed to have values since the ASP functions are only ever called at code-generation time, after all scalar and array assignments have been generated).

The argument template is a representation of the arguments to the function as supplied by the user. At run time this is matched to the actual parameters so that the appropriate substitutions between the user's dummy variables and the NAG parameters can be performed. The argument template is a list each of whose elements is either an identifier or a list whose car is an identifier and whose cdr is a number or irena variable. So a function:

```
foo(x1,x2,x3,delta) = ...
```

will match an argument template *'((x n) eps)* where the value of n is 3.

We will now describe the general cases:

6.2.4 Functions.

This is the common case where the routine returns a function value or a set of function values. The template is called "functions.tem", and the necessary values are:

function-dimension	The number of functions
output-name	The FORTRAN parameter which returns the results
argument-template	The argument template

6.2.5 Jacobians.

This is the case where the routine returns either a derivative or a Jacobian matrix. The template is called "jacobians.tem", and the necessary values are:

function-dimension	The number of functions
jacobian-dimension	The number of independent variables
jacobian-variable	The independent variable (which may be subscripted)
input-name	The name of the function (set) being differentiated
output-name	The FORTRAN parameter which returns the results
argument-template	The argument template

6.2.6 Functions and Jacobians.

Occasionally a routine returns both the value of the functions and their derivatives. There may be control statements in the ASP to control which values are returned. The expression *control1* is inserted before the function values are returned, *control2* between the function and jacobian values, and *control3* after the jacobian values. These expressions are all strings or lists of strings. This template is called "fun_and_jac.tem", and the necessary values are:

control1	The first control statement(s)
control2	The second control statement(s)
control3	The third control statement(s)
foutput-name	The FORTRAN parameter which returns the function values
joutput-name	The FORTRAN parameter which returns the derivatives
function-dimension	The number of functions
jacobian-dimension	The number of independent variables
jacobian-variable	The independent variable (which may be subscripted)
argument-template	The argument template

6.2.7 Hessians.

This is the case where the routine returns a Hessian matrix. The template is called "hessians.tem", and the necessary values are:

input-name	The name of the function (set) being differentiated
hessian-variable	The independent variable (which may be subscripted)
hessian-dimension	The number of independent variables
loutput-name	The FORTRAN parameter which returns the results in the upper triangle
doutput-name	The FORTRAN parameter which returns the results in the diagonal
argument-template	The argument template

6.2.8 Hessian Products.

These ASPs, used in a few optimisation routines, return the product of an array of functions and its hessian at a given point. The template is called "hess_products.tem", and the necessary values are:

input-name	The name of the function (set) being differentiated
hessian-variable	The independent variable (which may be subscripted)
hessian-dimension	The number of independent variables
residual-name	The name of the array containing the values of the functions at the given point
output-name	The FORTRAN parameter which returns the result

6.2.9 Dummies.

In this case the ASP function contains a list of statements which are inserted in the FORTRAN between the type declarations and the RETURN statement. Typically these might be an EXTERNAL statement and a call to a NAG routine. The template is called "dummies.tem" and the necessary value is:

literals	A list of any statements to be included

6.2.10 Matrix routines.

These are matrix manipulation routines which call a NAG routine to perform their function. The user may be required to provide a matrix. The template is called "mat_dummy.tem" and the necessary values are:

user-mat	The name of any matrix supplied by the user
nagfun	The name of the NAG routine being called (a string)
call-statement	The call to the NAG routine (a string)

6.2.11 Regions.

These are routines which calculate the boundaries of a region. The user provides a rectangle, The template is called "rectangles.tem" and the necessary values are:

low-out	The FORTRAN variable for the lower bounds
up-out	The FORTRAN variable for the upper bounds
dependents	A list of variables which the bounds may be expressions in (in the same syntax as an argument template)
rectangle-dimension	The dimension of the rectangle

6.3 Constructing special ASP templates.

Although most ASPs fall into the categories described in the previous section, some inevitably do not[2]. This section describes how to build individual templates for them, and details some of the relevant parts of the template mechanism.

ASP templates are normally generated in two passes through GENTRAN, as described in § 2.2. The first pass does most of the work, and may declare types using the GENTRAN *declare* function. The second will print out the FORTRAN type declarations, and may also output declarations for *irena constants* (see § 3.6.1). Each of these requires three components in different places: a type declaration, an `EXTERNAL` statement, and an assignment. The necessary statements to generate these are inserted in the template during the first pass by the three procedures *pass2-1, pass2-2, pass2-3.* The actual handling of the two passes is carried out by the two procedures *header* and *footer*, which should be respectively the first and last active statements in the template. The FORTRAN type declarations and header are generated by passing the ASP name to the procedure *asp-head.*

When expressions are translated, the system replaces the dummy parameters with their actual FORTRAN equivalents. It does this by temporarily assigning the value of the actual parameter to the dummies. If these are arrays then they must be declared as *operators* in Reduce. Any values which these parameters have must also be temporarily *cleared* since otherwise when we resimplify we will pick these values up, and conversely any values which the dummies have must be saved and restored later. The following procedures accomplish these tasks:

cleanse(l)	Temporarily clears each member of list l
prepare(u)	Temporarily makes identifier u an operator
restore(l)	Undoes the work of the previous procedures

There are procedures to accomplish this whole process for specific types of ASP — *prepare-names* and *prepare-rectangle-names* — which return lists to be passed to *restore* later on. The former is generally used for functions and Jacobians and takes three

[2]In the current system, out of just over seventy ASPs, these exceptions number only a dozen.

arguments: the ASP name, the name of the output parameter, and the expanded argument template. This can be generated by passing the argument template to the procedure *expand-at*, which converts e.g. *'(eps (x n) y)* to *'(eps (x 1) (x 2) (x 3) y)* where n has the value 3. The latter takes four parameters: the ASP name, the name of the variable for the lower bounds, the name of the variable for the upper bounds, and a list of the dependent variables.

The other two useful procedures are *generate-functions* and *generate-jacobians*, which produce the necessary assignments for functions and jacobians respectively. The former takes four parameters: the name of the output parameter, its first index (if appropriate), the name of the function (set), and the number of functions. The latter takes six parameters: the name of the output parameter, its first index, the name of the function (set), the name of the independent variable, the number of functions and the number of independent variables. The first index is passed so that the same output variable can be used to return the values of different functions (for example as in the SUBROUTINE G of D02RAF).

Finally there are two statements to allow the optimiser to be applied to sets of assignments. The first, which should occur before any assignments are generated, is:

```
IF !*GENTRANOPT THEN <<!*OPTSTACK!* := NIL;ON OPTIMISEWAIT>>$
```

and the second, which actually triggers the optimiser and causes output to be produced, is:

```
IF !*GENTRANOPT THEN OPTIMISE!-STACK()$
```

6.4 Summary

We have described our choice of representations for ASPs, and some of the philosophy behind them. We have also shown how most ASPs are instances of certain classes, and described the extensible descriptor language which we have designed to make their generation easier.

Chapter 7

How IRENA Works.

So far we have given an overview of IRENA from the user's point of view, and described how we transform the objects in our interface into the FORTRAN objects required by the NAG Library routines. In this Chapter we will describe how IRENA actually writes FORTRAN programs, and how the resulting code is linked to Reduce.

7.1 The Information Files.

Before doing this we must describe one more component of the system. The jazz and defaults files describe the IRENA interface, but we need a description of the FORTRAN interface as well. This is provided by the *information* file, which contains:

- the types of all NAG parameters;
- whether parameters are used for input, output, workspace etc.;
- which of them are arrays, and their dimensions;
- a list of ASPs and their types;
- the list of diagnostics to be printed if a NAG error occurs.

7.2 Generating The Code.

The initial sequence of steps taken by IRENA is as follows:

- The routine's information file is processed.
- The routine's *jazz* file is processed.
- Any *alias* file is processed.
- The key list is parsed and, where appropriate, values are assigned.
- The defaults files are processed: first the user's and then the system's.
- A check is made to ensure that all the parameters have values. At this point IRENA may look at global REDUCE values or prompt the user for values, depending on the settings of the switches ENVSEARCH and PROMPTVAL.

At this point, assuming that all necessary values have been provided, we are ready to proceed. In outline, what we do is as follows: we generate a FORTRAN program to call the NAG routine, this code is compiled and linked, and then made part of the running LISP system. The resulting subprogram is then called, being passed pieces of memory in which to deposit its results. On exit, these pieces of memory are examined and, if necessary, their contents transformed to produce the output structures for the user.

We are fortunate that PSL allows us to link a piece of foreign code into the running LISP system, as this facility is not generally available. Unfortunately there are difficulties in calling a *foreign function* from PSL. There is a limit of five parameters, while NAG routines often return many more. The easiest way round this problem is to pass the generated FORTRAN routine a block of memory in which to deposit its results, but FORTRAN cannot (legally) manipulate pointers. Thus we use a small C program as a form of "syntactic glue" between the LISP and the FORTRAN. It takes a block of memory as its argument, and calls the FORTRAN with a sequence of pointers to parts of this array. In practice we use this block of memory for workspace as well as output parameters, to reduce the size of the compiled FORTRAN code. This memory is allocated on the PSL heap so that, if we have to allocate a larger piece than usual, the old piece can be reclaimed (the heap is garbage-collected).

This C is useful for another purpose, since we can incorporate an error handler to detect the standard floating point (IEEE) exceptions such as division by zero, overflow etc. This allows us to exit gracefully and display some form of explanatory message, should an exception arise. In future, more NAG routines may incorporate advice on what to do in cases such as overflow, which we could display in the same way as we currently deal with non-zero IFAILs.

Thus we generate two pieces of code, one in C and one in FORTRAN, using the template processing facilities of Gentran. Hence each NAG routine has two Gentran templates associated with it — a C one and a FORTRAN one.

7.3 Loading the compiled code

The compiled code is linked into the running LISP system using a PSL utility called *oload*, which makes the foreign code part of PSL's binary program space (BPS). BPS is organised so that data grows down from the top while text grows up from the bottom[1] and, as there is no garbage collection done on it, when the text and data portions collide it is irrevocably full. There are two components of *oload*: one a unix shell script; the other a set of Lisp functions. Calling the Lisp function *oload* spawns a child process to execute the script which, given one or more compiled files, does the following:

- It generates a file containing the text portion of the file(s) being *oload*ed.
- It generates a file containing the data portion of the file(s) being *oload*ed.
- It creates a file containing the memory locations at which to load these files.
- It creates a file of Lisp statements telling the PSL compiler the names and entry points of the *oload*ed functions.
- If requested, it creates a new symbol table, so that the user may produce a new PSL with the *oload*ed code as an integral part.

[1]Technically the data doesn't go into BPS but into the word array space but, in PSL, these are both parts of the same structure, so the distinction can be ignored.

The LISP function *oload* takes as argument a string containing the names of the executable load modules and various options either for the *oload* script or *ld*, the Unix linker. The *oload* options control how much the user is told about the progress of the *oload*, whether a symbol table is produced etc. The ld options tell it which libraries to scan, where to find them and so forth. The script also receives the initial addresses of the end of the text segment and the beginning of the data segment.

The difficulty is caused by having to arrange matters so that the data segment of the new code *ends* at the *beginning* of the current data area of BPS. *Oload* does this through a tortuous sequence of steps:

1. using *ld*, it creates a relocatable code file containing all the additional modules;
2. using the Unix pattern scanner *awk*, it determines the relative sizes of the two segments. At this point it checks to see if BPS is going to be exhausted and, if so, terminates;
3. using these figures it works out how much bigger the text segment would need to be to "shunt" the data forward so that it was contiguous with the current data area;
4. it creates a blank piece of text this size by creating a file of null characters and then assembling it;
5. it then does a second *ld*, this time including the blank text, to create the final executable file;
6. this file is then split in two using *dd*, the "truncated" text (i.e. ignoring the blanks) and the data;
7. using the Unix utility *nm* the list of names and entry points is created;
8. if required, a further *ld* is performed to generate the symbol table.

The Lisp routine then reads in the files telling it where to put the pieces of code and loads them at the appropriate places.

There are three problems with this approach. The first one is that the whole process can be agonisingly slow. Up to three passes of the linker, the *awk* scripts and, worst of all, writing millions of zeros onto disc[2] twice can take a long time. The second problem is that the user needs to have enough free disc space for these big temporary files (there are two — one unassembled and one assembled) and, in a distributed networking environment, there can be problems with connections timing out. The third problem is to do with the way we are using *oload*. Each time we *oload* a routine we load all the NAG library routines it calls along with it and so BPS becomes rapidly exhausted. Enlarging BPS was found to exacerbate the first two problems to an almost unbearable degree.

In fact, *oload* does far more than IRENA needs. We never call a routine more than once, so we do not need to preserve it in BPS. Since we never "dump out" our system we do not need to produce a new symbol table, or even align the text and data segments correctly. Thus we have created our own version of *oload*, called *IRENAoload*, which is designed to be used for our "one-shot" problems. We need only one pass of the linker, and the text and data portions are placed in memory exactly as they would be by the loader. The pointers to the top and bottom of BPS are unchanged so that, the next time anything is loaded into BPS, the foreign function is overwritten. This method is much faster, doesn't require lots of temporary disc space and doesn't exhaust BPS.

7.4 Efficiency

Every time the user calls a NAG routine with IRENA, the whole process of compiling and oloading is gone through. Once the routine has been called, all this new code is effectively forgotten. In the case where a user is making multiple calls to a routine, perhaps changing one parameter each time, this may appear extremely inefficient. This inefficiency can be justified for the following reasons:

- The NAG Library is large, about 7Mb at mark 13, and so could not be loaded in its entirety in any practical situation. Individual NAG routines are also quite

[2]In the author's standard IRENA system, the gap in the middle of BPS starts at well over a megabyte.

large, and normally call many auxiliary routines. If the user were calling several different routines, oloading those Library routines required each time, then Reduce would run out of BPS very rapidly — experiments showed that the number of IRENA calls that could be made was in single figures. (BPS in PSL is not garbage collected.)

- There is a large constant factor in the time taken by IRENAoload, regardless of how much code is being processed. Even if the user were calling the same routine repeatedly, some recompilation would be required if:
  - Any of the parameters which changed were ASPs.
  - Any matrix parameters changed in size, since this would necessitate changes in the location of the pointers passed by the C code.
- It is not always trivial to pass Reduce algebraic objects to compiled foreign code, since they may be domain elements. For example, if the user sets the precision of rounded numbers to the machine precision, then the resulting objects will not be machine floats but will be bigfloats, represented by pairs of numbers: a string of integers and an exponent.
- The order of matrices being passed from PSL to FORTRAN and back must be reversed since PSL is implemented in C and so stores its arrays in row order, whereas FORTRAN stores them in column order.
- Values involving machine constants (see § 3.6.1) are generated by calls to the relevant NAG routine. Generating them separately in the Reduce environment to pass to a NAG routine, rather than producing the calls in the oloaded code, could spark more oloads (though in this case the obvious thing to do would be to duplicate their functionality with a set of LISP functions).
- We have tried to make our code the most general possible. If all parameters were passed, rather than assignment statements being generated as at present, it might lose some of its utility.

Thus we believe that, although sometimes inefficient, this mechanism is the most generally applicable. There are nevertheless improvements which could be made to this process. One possibility would be to write our own linker, which would directly generate the objects we require, rather than forming one big file with *ld* and subsequently splitting it up. Another possible improvement would be to allow loops to be incorporated in the generated code, with returned objects gaining an extra dimension to incorporate the series of results. So for example an output parameter which was normally a scalar would become a vector, a vector would become an array, an array a three-dimensional structure and so on. This would deal with the case where we are calling the same routine repeatedly, e.g. starting an optimisation from different points to ensure we get a global rather than local answer.

Given that we are using a one-shot technique, it must be asked whether it is worth *oload*ing at all. Why not fork off a subprocess which could write its results to a file which could then be read by Reduce? Though superficially attractive, there are some difficulties with this approach:

- We would normally end up generating very large files of data, most of which is uninteresting (e.g. workspace arrays whose first few elements need to be returned to the user). This could be avoided by doing some or all of the output jazzing process in the FORTRAN, but this would reduce the utility and readability of the generated code for other purposes.

- Straight conversion of floating point data from binary to ascii and back again will often cause that data to be corrupted, though encoding algorithms do exist to avoid this.

7.5 Operating System Dependencies

Clearly we are dependent on certain features of the operating system, and of the underlying LISP. The most obvious of these is the ability to fork off child processes to perform the compilations and run the oload script.

IRENA implementations have been made for SUN3 and SUN4 workstations, running

the SunOS 3.4, 4.0 and 4.1 operating systems. The differences between implementations on the two architectures are fairly trivial — a few changes to the oload script to reflect the differing segment sizes, and different arguments to the compiler[3]. Each change of operating system, however, has been hampered by bugs and incompatibilities in the linker, and changes in the organisation of some of the system libraries. The extreme fragility of this part of the process is another argument in favour of developing our own version of *ld*.

[3]SUN3's have various different ways of doing floating point arithmetic depending on what sort of co-processor they are fitted with, while the SUN4 (SPARC) architecture hides all the details from the user.

Chapter 8

The Design and Construction of the IRENA System.

So far we have described the IRENA system, but not the rationale behind its design. Describing that rationale is the purpose of this chapter.

The IRENA system consists of interfaces to virtually every routine in the NAG Library (the exceptions are those few routines which use the *reverse communication* technique[1]). Each interface has three essential components and two optional ones. The essential components are:

- The information file.
- The FORTRAN template.
- The C template.

These define the mechanism which IRENA uses to call the NAG routine. There are then the two optional components which transform the interface:

[1]The reverse communication technique involves the routine requesting data from the calling program at intermediate stages. The usual mechanism is that the routine is called a large number of times, the intermediate calls having only a few parameter values changed by the user. This gives an expert programmer greater flexibility since he or she may monitor the progress of the computation and control how the algorithm operates. However we feel that this model is inappropriate for IRENA, since it requires deep knowledge and understanding of the underlying algorithms. In any case there are only eight such routines in the Library.

- The system defaults file.
- The jazz file.

Additionally there are three utilities which may need to be extended to handle the interface to a newly-introduced routine:

- The ASP system.
- The jazz-functions.
- The out-functions.

The first of these is essential; the others, being part of the jazz system, are optional.

8.1 The Evolution of the NAG Library.

As stated in § 1.1.1, a new mark of the NAG Library is released roughly every eighteen months. At this point new routines are added and old, obsolete ones withdrawn. For some existing routines, additional information about their parameters is supplied to the user (for example the contents of the workspace array `W` mentioned in chapters 3 and 5 were not documented until the release of mark 13, even though the routine had been part of the Library since mark 8). Thus for IRENA to remain in step with the NAG Library at each new mark it will be necessary to scrap some old interfaces, identify those which have changed and modify them, and create many new ones. As an idea of the scale of this task, at mark 14 seventeen routines were withdrawn, whilst roughly one hundred and fifty new ones were introduced, giving a total of nearly one thousand routines in all.

8.2 The Evolution of Reduce.

A new version of Reduce appears roughly every two to three years, though new packages and upgrades of existing ones are available by electronic mail from the main development site (the RAND Corporation, Santa Monica) via an automatic server. Changes from version to version can be fairly major, but they are available well in advance to the beta

test sites (of which Bath is one). The change between Reduce 3.3 and 3.4 which had the most impact on IRENA was the new floating point system, which necessitated changes to Gentran as well. Minor changes to the parser, and in the way matrices are handled, also had some effect on us. Changes to Reduce will tend to be integrated with IRENA as they happen, and so pose less of a problem than changes to the Library.

8.3 Changes in the operating system.

These affect the dynamic linking process. As mentioned in chapter 7 the main problem so far has been with bugs in the unix linker. The other things which affect us are the compilers. On the SUN 4, the introduction of a new FORTRAN compiler requires that a fairly trivial change be made to the compiler arguments IRENA uses. There are also problems with the representation of double precision complex arrays.

8.4 Generating the interfaces.

It should be clear that, while the host system and the system code are fairly stable, if we wish to keep in step with the NAG Library we need to automate as much of the production of the interfaces as is feasible. To do this we have developed a two-phase process which generates the essential parts of the interface: the information files and the templates. This process uses a suite of programs which, together, comprise an interface compiler. The first program creates an abstract classification of the routine, as described in chapter 9. If this classification is not perfect, then it may be hand-modified at this stage. The classification is based on NAGs own documentation. From this the information file and the templates are generated completely automatically.

8.4.1 Defaults

At present the defaults files must be prepared by hand. However, based on the current documentation and the work we have done in this area, NAG have begun to include constraints and recommended values in their documentation in a regular, consistent way. While these will not cover all eventualities, as it is not really sensible to recommend

values for some tolerances or diagnostic print parameters; it would at least be possible to generate automatically skeletal defaults files which would include array dimensions and workspace. This would improve our automatically generated interface a great deal.

8.4.2 Jazzing

The design of a jazzed interface is, essentially, a subjective matter, and so must be done by hand. This may involve writing new jazz-functions and out-functions, and even altering the jazzing of existing routines to maintain consistency.

8.4.3 ASPs

Our automatically-generated interfaces do the job in all but one respect. Routines with no ASPs present no difficulties, and an ASP which is equivalent to one already encountered in the Library can be handled. However if a new kind of ASP is discovered then the ASP mechanism must be extended to handle it.

In the earlier versions of IRENA, each kind of ASP had its own template, but as the number of identified types grew we found this system to be grossly unsatisfactory. Thus we developed the "descriptor language" described in § 6.2, to make updating the system as quick and as error-free as possible. Even the task of hand-crafting a template for a "maverick" ASP has been made relatively painless.

8.4.4 Documentation

We would like the generation of the documentation for each IRENA routine to be completely automatic but, because of the stylistic considerations involved, this is not possible. However, we are developing a tool which will use the specification, jazz and defaults files, as well as the NAG documentation, to produce rough routine documents which can then be polished by hand.

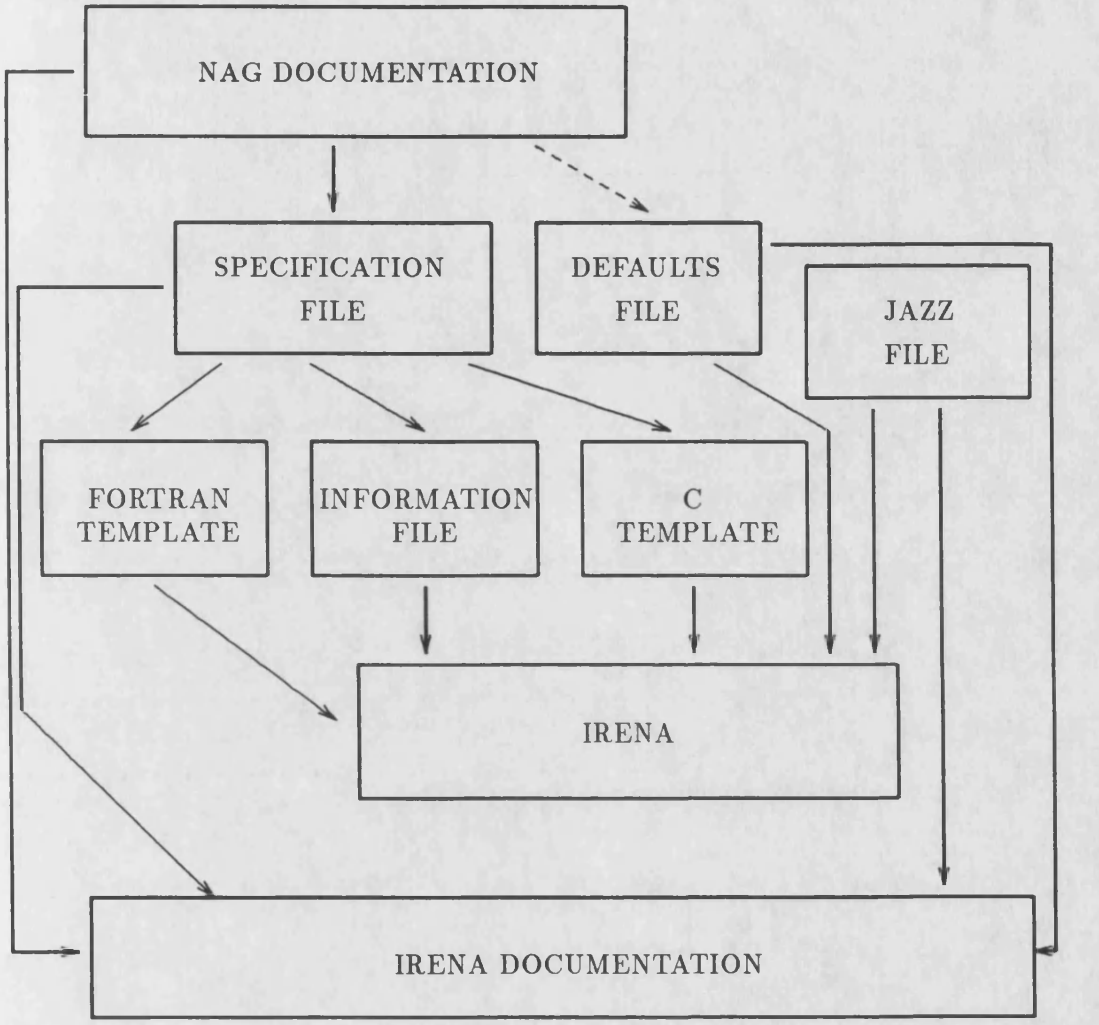

Figure 8-1: Schematic view of the generation of the IRENA interfaces.

8.5 Summary

Figure 8-1 gives a schematic view of where the components of the interface come from. When a new mark of the library is released it is necessary to:

1. Classify all the routines in the new mark using the classify program described in chapter 9.
2. Update the ASP system if necessary (this is clearly indicated by the routine classifications).
3. Automatically generate the relevant components of the interfaces.
4. Compare the existing classifications of routines with their new ones to see if their description has changed (e.g. using *diff*), and if necessary revise their jazzing and defaults.
5. Write jazz and defaults files for new routines.
6. Prepare the new documentation.

By automating the basic interface-generating process, and by using high-level descriptions for ASPs and jazzing, we are able to produce reliable, error-free interfaces much quicker than would otherwise be the case. This will allow us to maintain IRENA in step with the NAG Library.

Chapter 9

Classifying NAG routines

As has been stated previously, several components of the IRENA system are generated automatically, namely the program templates and information files. In this chapter we will describe how we prepare the data which the generating software requires. Taken together, this comprises a classification of the functional aspects of the routine. The information we require is:

The routine type i.e. SUBROUTINE, //real//[1] FUNCTION etc.

The argument list A list of the routine's arguments in the order in which they should be provided to the routine.

The argument types For each parameter we need to know its type, and whether it is an array. In the latter case we also need to know its dimension(s).

How the arguments are used We need to know if a parameter is used to provide information to the routine (an *input* parameter), to return results from the routine (an *output* parameter), for both (an *input/output* parameter), purely as *workspace*[2], or is simply a *dummy* parameter.

[1]The slashes have their usual NAG meaning — that the name they enclose is implementation dependent. In this case they mean that the actual type depends on the precision of the implementation.

[2]We insist that workspace arrays have no other use, i.e if an array is used partly as workspace, and partly for output, then we ignore the former.

Details of subprogram parameters We need details about ASPs including their parameters, parameter types, and origin (i.e. whether provided by the user or the NAG library).

Diagnostic information We need a list of messages to display should we encounter a non-zero `IFAIL` or any other error on exit from a routine.

To generate this by hand would be both tedious and error-prone, so we attempt to extract automatically as much information as we can from the NAG online documentation.

9.1 The NAG Help program

NAG have an online version of their manual, which can be interrogated using a descendant of the program described in [Hazel & O'Donohue 1980]. We are only really interested in the data files. During the course of this project both the program and the contents of the data files have undergone major revisions. In particular, information which we have identified as useful (such as whether a parameter is used for input, output etc.) has been explicitly incorporated. This chapter describes the mark 12 and 13 system.

The entry for each routine in the NAG help files is split up into the following *members*:

§A A description of the problem which the routine is designed to solve;

§B The routine heading with the type declarations "commented-out";

§C A description of every parameter, each of which is in its own *sub-member*[3];

§D The error diagnostic messages.

Each member and sub-member is preceded and succeeded by one or more *directive lines* which serve to signal its start and end points, as well as giving an indication of which

[3]occasionally several related parameters are grouped together into one sub-member.

member follows and, in the case of the sub-members, the name of the parameter (in slightly truncated form). These directive lines are the only reliable structure to the Help files, they allow us to determine the correct piece of text to process at any given time, but nothing else.

The Help program works in a fairly straightforward manner. It parses the input line supplied by the user looking for a key which exists in its index. It then finds the relevant member and, after extracting any remaining keys from the input line, scans through it displaying the appropriate sub-members. These are listed verbatim on the screen. What the user sees is precisely what is present in the Help files, less the directive lines. Unfortunately the data files have evolved over the years, been added to by different people, and so (globally) there is no reliable structure or layout. On the other hand the contributors have tended to follow a similar sort of pattern in the way they have formatted the files, and so it is often possible to make assumptions about text layout in a *local* context. Had we been dealing with text which was designed to be processed before it was viewed, then the layout would have been better standardised, since a contributor would be forced, for example, to use the system-defined tabs, rather than whatever he or she felt looked good at the time[4]. Unfortunately, when we started the IRENA project, the printed manual had not been fully typeset, otherwise we could have used that.

Despite the fact that the NAG Help program has been in use for many years at a great many sites, the data files were found to contain quite a large number of errors. Most of these would at their very worst have proved a minor inconvenience to an (English-speaking) user. Many were no more than minor spelling mistakes or minor "syntax-errors" in §B. For example in:

```
      SUBROUTINE  //D01AMF//  (F,  BOUND,  INF,  EPSABS,
     1   EPSREL,RESULT, ABSERR, W, LW, IW, LIW, IFAIL)
C     INTEGER INF, LW, IW(LIW), LIW, IFAIL
C     //real//  F, BOUND, EPSABS, EPSREL, RESULT, ABSERR
```

[4]For example the UNIX online manual is passed through the *nroff* text-processor before being displayed, and is formatted by its own special package of macros.

```
C     1 W(LW)
C      EXTERNAL F
```

the comma is missing at the end of the fourth line. This might seem like a trivial mistake (and indeed, from the point of view of an ordinary user of the Help system it is), but it is the sort of thing we might rely on to signal the presence of a continuation line, rather than wait for the continuation card. Another example, which is a little more difficult to pin down, is the misspelling of OUTPUT in the last line of:

```
      SUBROUTINE //D02BBF// (X, XEND, N, Y, TOL,
     1   IRELAB, FCN, OUTPUT, W, IFAIL)
C     INTEGER N, IRELAB, IFAIL
C     //real// X, XEND, Y(N), TOL, W(N,7)
C     EXTERNAL FCN, OUPUT
```

The lesson to learn from cases like these is that it is unsafe to make any assumptions about either the overall correctness of the text or the pieces of code given in §B. An automatic spelling checker has been employed, but obviously this would not have helped in either of the cases shown above (OUPUT is after all a pretty reasonable parameter name). This does however mean that we can rely on the correct spelling of any "normal" English words which we want to spot, such as "function" or "variable" etc. In one case a sub-member had had the multiple occurrence of a phrase removed, so that it made absolutely no sense whatsoever. It is also worth mentioning that the names of parameters in different routines in the library generally bear no relationship to each other, and are fairly inconsistent. The only exceptions to this rule are many of the workspace parameters which have names like WORK, IWORK etc., their dimensioning arguments which tend to be called LWORK, LIWORK etc., and the diagnostic parameter which is always called IFAIL.

9.2 General Strategy

We can extract quite a lot of the information we need from §B, namely the routine's type, argument list, and argument types. The diagnostic information can be extracted from

§D. This leaves us with details of subprogram parameters to extract, and information about ASPs. We will leave a discussion of the latter until the next section, and concentrate here on scalar parameters.

The strategy we follow is basically to search for certain carefully chosen keywords or strings, and apply a set of simple rules. This works better than one might at first imagine because of the amount of jargon used in the Help files: these phrases may be regarded as having little or no ambiguity in the context of the NAG documentation, as opposed to in their normal English usage. The strategy is aided by the structure of the Help files described above: because they are designed to be read in small pieces we may identify a relevant section of text quite precisely, and not worry about the wider context. So for example, we may search for the string "UNCHANGED ON EXIT" to signal that a parameter is *not* an output parameter.

This naïve approach is too simplistic for two reasons. The first problem is that phrases may be permuted; the above case might also appear as "ON EXIT X IS UNCHANGED", or "THE CONTENTS ON EXIT ARE UNCHANGED". So in some cases we need to identify the active components of a phrase, which here are "EXIT" and "UNCHANGED", and search for them without strict rules about ordering or position. Thus we need to be able to define areas of text in which a number of such words or phrases are liable to refer to one another. To do this we introduce the notions of *periods* and *semantic groups*. A *period* is a piece of text delimited by the characters ".", "?", and "!" (for completeness we can pretend that there is an invisible full-stop at the start of each section of text we process). A *semantic group* is a piece of text within a period which is additionally delimited by the characters ",", ";", or ":". We then try to match each occurrence of "unchanged", with the right occurrence of "entry" or "exit", by connecting couples which occur within the same semantic group. If no matching word is found then we look in the wider context provided by its period. If we still do not find a match then we ignore it, and if we find a conflict, then we mark the parameter as unclassifiable.

The other problem with the naïve approach is that the immediate context of the word or phrase may actually contain a modifier which alters or negates its normal meaning.

For example the phrase "X IS NORMALLY UNCHANGED ON EXIT" implies that (in certain situations) the parameter *is* used for output. Thus we have to take account of the presence of *modifiers*, which we do by ignoring the word or phrase which follows them, unless they are immediately succeeded by a punctuation mark. The primary phrases we look for, with the flags which their presence sets, are given in table 9.1:

Phrase	Flag
"ENTRY"	entry_found
"INPUT"	entry_found
"USER MUST SET"	entry_found
"EXIT"	exit_found
"RETURN"	exit_found
"RETURNS"	exit_found
"UNCHANGED"	unchanged_flag
"UNDEFINED"	unchanged_flag
"INDETERMINATE"	unchanged_flag
"ARRAY IS OVERWRITTEN"[5]	unchanged_flag
"WORKSPACE"	workspace_flag
"WORKING SPACE"	workspace_flag
"WORKING STORAGE"	workspace_flag
"WORK SPACE"	workspace_flag
"WORK STORAGE"	workspace_flag
"DUMMY VARIABLE"	dummy_flag
"DUMMY PARAMETER"	dummy_flag
"DUMMY ARGUMENT"	dummy_flag
"ARRAY DIMENSION"	array_flag
"ARRAY OF DIMENSION"	array_flag

Table 9.1: Primary phrases recognised by the classify program.

At the end of each semantic-group, period or sub-member, we apply the following set of rules:

- If we have **entry_found** then we try to decide whether it is genuine, or part of an "unchanged on exit" type phrase. We also ignore it if we have **exit_flag** since we have found that entry and exit descriptions always occur in order, and so if found

[5]This may seem a bizarre phrase to equate with "UNCHANGED". In fact it normally indicates that the parameter is used both for input and workspace. Thus the phrase "on exit, the array is overwritten" does not imply that it has been overwritten with anything interesting.

together the entry part is spurious. The use of **local_entry_flag**, which persists over a whole period, is explained below. So the rule is:

if **entry_found** *then*
  if ~**unchanged_flag** ∧ ~**exit_flag** *then*
    entry_flag = **true**
    local_entry_flag = **true**
  else
    unchanged_flag = **false**
  entry_found = **false**

- If we have **exit_found** then the procedure is similar to the first case:

if **exit_found** *then*
  if ~**unchanged_flag** *then*
    exit_flag = **true**
    local_exit_flag = **true**
  else
    unchanged_flag = **false**
    unchanged_on_exit_found = **false**
  exit_found = **false**

- If we find we have a lone "unchanged", then we look beyond the current semantic group to the whole period. Here we are dealing with phrases like "ON EXIT, X IS UNCHANGED". This is where we use **local_entry_flag** and **local_exit_flag**. Note that we ignore the possibility of a phrase like "ON ENTRY, X IS UNCHANGED ...". In fact, if this sort of phrase occurs, it tends to be of the form: "ON ENTRY, X IS UNCHANGED FROM THE PREVIOUS CALL ..." which means that we *are* dealing with an input parameter (albeit of a rather special kind).

if **unchanged_flag** *then*

if **local_entry_flag** $\wedge$ $\sim$**local_exit_flag** *then*

entry_flag = **false**

else if $\sim$**local_entry_flag** $\wedge$ **local_exit_flag** *then*

exit_flag = **false**

else if **local_entry_flag** $\wedge$ **local_exit_flag** *then*

ambiguous_flag = **true**

- Finally, if we are at the end of a period, we must reset the **local_entry_flag** and **local_exit_flag** values.

if **period** *then*

local_entry_flag = **false**

local_exit_flag = **false**

At the end of each sub-member we apply the following two rules:

if **array_flag** *then*

the parameter is an array.

if **ambiguous_flag** *then*

the parameter cannot be classified

else if **exit_flag** $\wedge$ **entry_flag** *then*

the parameter is an input/output parameter

else if **entry_flag** *then*

the parameter is an input parameter

else if **exit_flag** *then*

the parameter is an output parameter

else if **dummy_flag** *then*

the parameter is a dummy parameter

else if **workspace_flag** $\wedge$ **array_flag** *then*

the parameter is a workspace array

else if **array_flag** $\wedge$ **"WORK"** $\subset$ parameter name *then*

the parameter is a workspace array

else if parameter name of form L*WORK *then*

the parameter is an input parameter (in fact an array length)

else if **unchanged_on_exit_found** *then*

the parameter is an input parameter

else

the parameter cannot be classified.

Clearly the further down this clause we get, the more desperate we are becoming!

9.2.1 Choosing the key phrases and rules.

This section briefly outlines how and why we chose the set of phrases and rules for reasoning about them listed above. We started by looking at just one chapter of the library (D01, the quadrature chapter), worked out our basic approach, and iterated on that. When we found a parameter which our system couldn't cope with, we tried to introduce a method to handle it. When we found a parameter which our system classified wrongly, we tried to alter our scheme to accommodate it. Absolute accuracy was not essential, as any incorrect classifications showed up in testing of the individual IRENA interfaces, and the specification files could then be modified by hand without disturbing the rest of the generation process. Nevertheless this approach turned out to be very successful.

There are three main categories of parameter which our system fails to classify:

1. *Probe* parameters which are used to pass information to ASPs from the user's program.

2. *Communication* parameters which are used to pass information between NAG routines.

3. Parameters whose description in the Help files is simply a pointer to another sub-member.

A more sophisticated program might have been able to follow the indirection which

causes the problem in the last case, but there are so few such instances that the effort didn't seem to be worthwhile.

9.3 ASPs

In chapter 6 we described how ASPs are handles, and how we classify their different types. Thus when we encounter an ASP in the NAG documentation, we need to be able to decide whether it is one which we've come across before. We can easily extract the ASP equivalent of §B of a routine member from the Help file, and of course we can "read through" the description in the sub-member.

There are two levels at which we classify ASPs. The first is based on their functional form, i.e.:

- Their type: SUBROUTINE, //real// FUNCTION etc.;
- The number of parameters they take;
- The types (including "arrayness") of the parameters.

The second level on which we classify ASPs is their actual purpose, e.g. return an array of function values, compute the jacobian of a system of equations etc. Unfortunately there is very little consistency within the library: ASPs which do the same thing in different routines are frequently not compatible. This is the main reason why we check the types and the order of the parameters as well.

9.3.1 Performing the classification

We spot that a parameter is an ASP by checking for the occurrence of the upper-case words "SUBROUTINE" or "FUNCTION" in its sub-member. We also have to check that the parameter is not an array, since they sometimes occur in the descriptions of *probe* parameters. Obviously if a parameter is an array it cannot be a subprogram, so there is no ambiguity here. ASPs are classified as follows:

1. When the classify program starts up, a file of data about all known ASPs (called *subprogram_params*) is read in and processed. It contains details of each type's

functional form, as well as a (possibly empty) list of identifying phrases to look for in the Help file.

2. When an ASP is encountered in the Help file it is classified according to the set of "functional" criteria described above. If there are no matching types in subprogram_params then it is classed as UNKNOWN.

3. We now have a list of one or more types to which our new ASP might belong. We then read through the sub-member looking for the phrases which match those of these candidates. If, at the end of this process, exactly one type matches exactly (i.e. all the phrases associated with it occur in the current sub-member) then we assign this type to our ASP. Otherwise we say that the type is UNCERTAIN, and we list the possibilities.

9.3.2 The subprogram data file

The data file has at least one entry for each type of ASP. Each entry has the following format:

- A line containing the type number.
- A line containing 1 if it is a FUNCTION and 0 if it is a SUBROUTINE.
- A line containing the number of arguments n.
- n lines containing the types of the arguments. An "@" character at the end denotes an array, so a real array has type "real@"[6].
- Zero or more lines containing (case insensitive) phrases to identify that type.

Entries are delimited by the string "****". We use multiple entries to give alternate sets of key phrases for the same type. A typical example of an entry in the data file is:

```
6
0
```

[6]Types are generic, not precision dependent.

```
4
integer
real@
real@
integer
must calculate the values of the functions at
and return these in the vector
****
```

which can be interpreted as meaning that type 6 ASPs are subroutines which take four parameters, the first and last of which are integers, while the others are real arrays. In addition their parameter description will contain the two phrases "must calculate the values of the functions at" and "and return these in the vector".

The strategy for identifying new ASP types and choosing a suitable set of key phrases will be discussed later.

9.4 IFAILs

IFAIL is used in three different ways by the NAG library to pass diagnostic information to the user when an error occurs:

1. The value of IFAIL is an index to a unique diagnostic.
2. The value of IFAIL is an index to a diagnostic which may not be unique.
3. There is only one diagnostic, but the value of IFAIL conveys some extra information. For example in F02BJF the value of IFAIL represents the place where an error occurred.

§D of each routine member in the Help files consists of the following:

- A preamble.
- A sequence of descriptions of what the output values of IFAIL mean. Each description is preceded by a header line of the form:

<indentation> IFAIL <op> <value>

The subsequent text is indented further (but uniformly) than the header line.

- Extra information relating to some or all of the diagnostics interspersed amongst them, and indented to the same depth as the header lines.

We want to ignore the preamble, and we chose to ignore the extra information, since it is not clear when it will be relevant[7].

The thing which determines how each diagnostic will be handled is <op>. There are six possible cases as follows:

1. = or .EQ.
2. < or .LT.
3. > or .GT.
4. <= or .LE.
5. >= or .GE.
6. .NE.

Not all occur in the current Help files, but we include them all for completeness. The output of the classify program is either the single word "None." in the case where **IFAIL** is not used by the routine[8], or a sequence of *diagnostics*. Each diagnostic consists of one or more *instruction lines* followed by the text of the diagnostic message. Each instruction line is of the form:

#<operation>[<value>]

where <operation> is one of EQ, NE, LT, GT, LE, and NE (the FORTRAN relational operators without the surrounding full stops). The case when <value> does not occur is when we have only one diagnostic which refers to the value of **IFAIL**. In this case we have the instruction line "#EQ".

[7]In fact NAG are endeavouring to remove most of these pieces of text, and incorporate them in individual diagnostics where appropriate.

[8]It may still be included as a parameter for completeness however.

9.5 Using the Classify program

The classify program is written in C, and the following description assumes that it is being used under Unix. The syntax for use is:

classify <routine-name> [<output-file>]

If no <output-file> is specified then the specification is sent to the standard output. The normal convention is to store the specification in a file called <routine-name>.s.

If an error occurs then an appropriate message is printed. Often the message can be a little misleading — for example if a parameter is misspelt in §B of the Help files then the classify program will normally complain about a missing type declaration.

Assuming that there are no errors in the Help files, there are two possible problems which can occur. The first is that a parameter is unclassified, in which case it will appear as a member of a list headed "UNCLASSED". The second is when the type of an ASP cannot be identified. In this case its type will be given as "UNCERTAIN" or "UNKNOWN".

Specification files with unclassified parameters are generally modified It may be that the ASP has been supplied by the NAG Library, in which case we hand modify the specification file. Otherwise we do the following: first check to see if the ASP is of a known type, but classify failed to recognise it. This is the case when classify has said that the type is "UNCERTAIN", and listed the possibilities. If it is in fact of a previously encountered type then we should either modify the key phrases in that type's entry in the data file (see § 9.3.2), or create a new entry for it with the same functional information but different phrases.

Where we genuinely do have a new ASP we must create a new entry in the subprogram data file. Before we can call an IRENA routine with that ASP we must add it to the ASP system (see chapter 6).

In general, after making any change to the subprogram data file, the user should run classify over the Library again to see whether he or she has managed to resolve any other problems, or introduce new ambiguities.

9.5.1 Changing certain parameter names

On running classify the user will notice that NAG parameters with the names E and T have been replaced by parameters called EEE and TTT respectively. This is because e and t are reserved words in Reduce and so it would be inappropriate to use them as parameter names in IRENA. EEE and TTT will normally be *aliased* to something more appropriate (see § 5.1.1). This doesn't change the diagnostic messages, but this is a general problem with aliasing.

9.6 The Specification Files

These are human-readable ASCII files. An example, the specification of D01AJF, is given in figure 9–1.

9.7 The Classify Program for the Mark 14 Library.

As stated at the start of this chapter, NAG undertook a fundamental revision of their documentation at mark 14. One aim of this was to accommodate future projects which would almost certainly require the same kind of information as IRENA. As a result of this all the information concerning parameter use (input, output etc.) is now explicitly incorporated in the documentation, so any errors in classification of parameters encountered is due to errors in that source.

TYPE

SUBROUTINE

SPECIFICATION

```
      //D01AJF//(F,A,B,EPSABS,EPSREL,RESULT,
     1 ABSERR,W,LW,IW,LIW,IFAIL)
C     INTEGER LW,IW(LIW),LIW,IFAIL
C     //real// F,A,B,EPSABS,EPSREL,RESULT,
C    1 ABSERR,W(LW)
C     EXTERNAL F
```

PARAMETERS

**** INPUT PARAMETERS:

A
B
EPSABS
EPSREL
LW
LIW

**** OUTPUT PARAMETERS:

RESULT
ABSERR
W@
IW@

**** INPUT/OUTPUT PARAMETERS:

IFAIL

**** WORKSPACE PARAMETERS:

None.

**** DUMMY PARAMETERS:

None.

**** FUNCTIONS:

NAME: F
SUPPLIER: USER
TYPE: 1

```
    //real// FUNCTION F(X)
    //real//          X
```

**** SUBROUTINES:

None.

IFAIL VALUES

#EQ1

The maximum number of subdivisions allowed with the given workspace has been reached without the accuracy requirements being achieved. Look at the integrand in order to determine the integration difficulties. If the position of a local difficulty within the interval can be determined (e.g. a singularity of the integrand or its derivative, a peak, a discontinuity, etc.) you will probably gain from splitting up the interval at this point and calling the integrator on the subranges. If necessary, another integrator, which is designed for handling the type of difficulty involved, must be used. Alternatively, consider relaxing the accuracy requirements specified by EPSABS and EPSREL, or increasing the amount of workspace.

#EQ2

Roundoff error prevents the requested tolerance from being achieved. The error may be under-estimated. Consider relaxing the accuracy requirements specified by EPSABS and EPSREL, or increasing the amount of workspace.
Please note that divergence can occur with any non-zero value of IFAIL.

#EQ3

Extremely bad local integrand behaviour causes a very strong subdivision around one (or more) points of the interval. Look at the integrand in order to determine the integration difficulties. If the position of a local difficulty within the interval can be determined (e.g. a singularity of the integrand or its derivative, a peak, a discontinuity ...) you will probably gain from splitting up the interval at this point and

```
    calling the integrator on the subranges. If necessary, another
    integrator, which is designed for handling the type of
    difficulty involved, must be used. Alternatively, consider
    relaxing the accuracy requirements specified by EPSABS and
    EPSREL, or increasing the amount of workspace.
    Please note that divergence can occur with any non-zero value
    of IFAIL.
#EQ4
    The requested tolerance cannot be achieved, because the
    extrapolation does not increase the accuracy satisfactorily;
    the returned result is the best which can be obtained. Look at
    the integrand in order to determine the integration
    difficulties. If the position of a local difficulty within the
    interval can be determined (e.g. a singularity of the integrand
    or its derivative, a peak, a discontinuity ...) you will
    probably gain from splitting up the interval at this point and
    calling the integrator on the subranges. If necessary, another
    integrator, which is designed for handling the type of
    difficulty involved, must be used. Alternatively, consider
    relaxing the accuracy requirements specified by EPSABS and
    EPSREL, or increasing the amount of workspace.
    Please note that divergence can occur with any non-zero value
    of IFAIL.
#EQ5
    The integral is probably divergent, or slowly convergent.
    Please note that divergence can occur with any non-zero value
    of IFAIL.
#EQ6
    On entry, LW < 4,
    or        LIW < 1.
    Please note that divergence can occur with any non-zero value
    of IFAIL.
```

Figure 9–1: The specification file for D01AJF.

Chapter 10

Examples of Using IRENA.

In this chapter we shall look at some longer examples which show how IRENA may be used to solve real problems. We shall also demonstrate some more features of the system.

10.1 A steel rolling problem.

In [Gomez 1990] the author describes a set of equations which represent the behaviour of a hot strip rolling mill. Given various parameters describing the milling machinery and the thickness and tension of the strip at the start of the process, it is possible to determine the thickness of the strip at the outlet, as well as some details of the process that has taken place. The author was unable to use his computer algebra system (in this case Maple) to solve the problem, and so was forced to use a numerical technique. The method he chose was to use Maple to generate a complete FORTRAN program which would solve the equations using Newton's algorithm. Using IRENA, however, the process is much simpler. Figure 10-1 shows the equations and parameter values as a Reduce file. Figure 10-2 shows the IRENA session used to compute the solution using the routine C05PCF, which finds the solution of a system of nonlinear equations. (The NAG manual describes this routine as "comprehensive" as opposed to its "easy-to-use" counterpart C05PBF. In both cases the FORTRAN user must provide the jacobian of the system to be solved, which is calculated automatically by IRENA. It is interesting

```
% A set of equations which describe the behaviour of a steel mill.

exp1 := h2 - s - (f + a2 * (1 - exp(a3*f))) / a1$
exp2 := f - 1*k*gr*(pi*sqrt(h2/gr)*atan(sqrt(r))/2-pi*csi/4-log(hn/h2)+
        log(h1/h2)/2)+gr*csi*t1/h2$
exp3 := atan(phi*sqrt(gr/h2))-sqrt(h2/gr)*(pi*log(h2/h1)/4+sqrt(gr/h2)*
        atan(sqrt(r))-t1/k/1/h1 + t2/k/1/h2)/2$

r := (h1-h2)/h2$ % The radius of the roll
csi := sqrt((h1-h2)/gr)$
hn := h2+gr*phi^2$

a1 := 610$
a2 := 648$
a3 := -0.00247$
1 := 1250$ % The width of the stand.
k := 0.014$
gr := 360$
t1 := 12$ % The inlet tension.
t2 := 35$ % The outlet tension.
h1 := 24$ % The initial thickness of the strip.
s := 12$ % The tightness of the screw.

end$
```

Figure 10-1: Equations describing a steel mill.

to note that in IRENA both routines have identical calling sequences, though C05PCF does have more control parameters.) The final zero of the set of equations represents the milling force, the final thickness of the strip, and the neutral angle (the angle at the point of the roll where there is no sliding). We have done the computation twice with different starting points as a check on the solution, since C05PCF will only find the nearest zero to the starting point. In this case there should be only one meaningful solution. Since both the equations and the zero are available in the Reduce environment, it would be simple to substitute the latter back into the former as a further check.

10.2 Warm starts after errors.

Some NAG routines include a warm start facility. If the routine exceeds its maximum

```
2:  in "steel.red"$

3: c05pcf(fcn1(f,h2,phi)=exp1,
3:        fcn2(f,h2,phi)=exp2,
3:        fcn3(f,h2,phi)=exp3,
3:        vec start{1000,15,0} )$

{ZERO,RESIDUALS,FCALLS,JACCALLS,SCALEFACTORS,JACOBIANQ,JACOBIANR,

 QTRANSPOSEF}

4: zero;

[ 1363.676366832  ]
[                 ]
[ 15.261235300841 ]
[                 ]
[0.063202457582016]

5: c05pcf(fcn1(f,h2,phi)=exp1,
5:        fcn2(f,h2,phi)=exp2,
5:        fcn3(f,h2,phi)=exp3,
5:        vec start{500,10,0.1} )$

{ZERO,RESIDUALS,FCALLS,JACCALLS,SCALEFACTORS,JACOBIANQ,JACOBIANR,

 QTRANSPOSEF}

6: zero;

[ 1363.6763668355 ]
[                 ]
[ 15.261235300853 ]
[                 ]
[0.063202457582067]
```

Figure 10-2: The IRENA session needed to solve the steel mill problem.

```
2: d01eaf(fset f[i=1:4](w,x,y,z)=
2:             log(w + 2*x + 3*y + 4*z)*sin(i + w + 2*x + 3*y + 4*z),
2:        region= [0:1,0:1,0:1,0:1])$
 ** MAXCLS too small to obtain required accuracy
 ** ABNORMAL EXIT from NAG Library routine D01EAF: IFAIL =     1
 ** NAG soft failure - control returned
    MAXCLS was too small for //D01EAF// to obtain the required
    accuracy. The arrays FINEST and ABSEST respectively contain
    current estimates for the integrals and errors.
{WRKSTR,CALLS,ABSERRS,INTEGRALS}

3: calls;
57

4: d01eaf(reenter, maxcalls=10000, fset f[i=1:4](w,x,y,z)=
4:             log(w + 2*x + 3*y + 4*z)*sin(i + w + 2*x + 3*y + 4*z),
4:        region= [0:1,0:1,0:1,0:1])$
{WRKSTR,CALLS,ABSERRS,INTEGRALS}

5: calls;
6384

6: integrals;
[0.038349753477037]
[                  ]
[0.40117322966197 ]
[                  ]
[0.39515988860082 ]
[                  ]
[0.025838368333277]


7: abserrs;
[0.000045932793434718]
[                     ]
[0.000045958066759794]
[                     ]
[0.000040497855131689]
[                     ]
[0.000045094532826377]
```

Figure 10-3: Doing a warm start with IRENA.

number of iterations it preserves the contents of its workspace so that the user can, if desired, pick up where he or she left off in a subsequent call. Figure 10-3 gives an example of this, adapted from the one in the NAG manual. D01EAF computes approximates to the integrals of a set of similar functions, each defined over the same region. If the routine exceeds its maximum number of iterations, it returns its current workspace in the array `WRKSTR`. A subsequent call to the routine using this array, and with the parameter `MINCLS` set negative, will start up where it left off and proceed until the maximum iteration limit `MAXCLS` is reached. In the example we find that 57 iterations was inadequate, so in the follow-on have increased the maximum number to 10000, which succeeds. Signaling the start of a continuation is done using the keyword *reenter* in the key list, which sets up the various parameter values as required.

10.3 Multi-Routine interfaces.

Sometimes it is nice to have one interface able to call several routines (or several sequences of routines). For example the routine C06EAF calculates the discrete Fourier transform of a sequence of real data values. The result is returned in hermitian form which can be displayed in Reduce using the special operator *display!-hermitian.* C06EBF calculates the discrete Fourier transform of a hermetian sequence of complex data values. To calculate the inverse transform the call to C06EBF must be followed by a call to C06GBF. If the user provides the keyword *inverse* in the call to C06EBF this will be done automatically. The example in figure 10-4 shows how this can be done to restore the original sequence after a transform has been calculated. This example also shows the use of the *print-precision* command for changing the printed precision of real numbers.

```
2: print!-precision 6$

3: c06eaf(vec sequence {0.34907,0.54890,0.74776,0.94459,
                        1.13850,1.32850,1.51370})$

{TRANSFORM}

4: display!-hermitian transform;

2.48361

-0.265985 + 0.530898*I

-0.257682 + 0.202979*I

-0.256363 + 0.0580623*I

-0.256363 - 0.0580623*I

-0.257682 - 0.202979*I

-0.265985 - 0.530898*I

5: c06ebf(inverse,sequence=transform);

{TRANSFORM}

6: tp transform;

[0.34907  0.5489  0.74776  0.94459  1.1385  1.3285  1.5137]

7: print!-precision(-1)$
```

Figure 10-4: An example of a multi-routine interface.

Chapter 11

Routine Selection

So far we have concentrated on how we may improve the interfaces to individual NAG routines by abstracting the choice of algorithmic and housekeeping parameters away from the user. While this removes the need to worry about the details of a particular routine's operation, it still requires the user to decide which routine is the best for his or her particular problem. This is not in general an easy decision to make, and often requires some "higher-level" mathematical insight into the characteristics of the problem being solved. For example, a user wishing to solve a one-dimensional integral over a finite region needs to decide whether the integrand is smooth, identify singularities and discontinuities, and so forth. Often, of course, he or she will simply choose the most general routine available, which might not actually be the best.

NAG tackles this problem in their documentation by providing both a general description of the problem domain, and advice on which characteristics should influence the choice of which routine (often as an easy-to-follow flow diagram). Unfortunately many phrases are vague or unexplained: for example in the flow chart for one-dimensional finite quadrature the user is asked whether his or her integrand is "fairly smooth". There is apparently no definition of the difference between a smooth and fairly smooth function.

Several interactive systems have been developed to aid the user in choosing an appropriate routine. The first such package was NAXPERT [Schulze & Cryer 1986, Schulze & Cryer 1988] which essentially automates the flow-charts in the manual. When

a user consults NAXPERT, he or she provides keywords to the system which describe the problem to be solved. The system will ask questions of the form *is the following keyword appropriate to your problem?* if necessary, and will eventually come up with a list of possible routines. If required, the system will print out a template program, which the user can then modify for his or her own purpose. Additionally, some extra information is available to explain the meanings of certain keywords.

There are also some similar but more general packages available. NITPACK [Gaffney & Wooten 1983] is a package which takes a representation of a decision tree and guides the user through it interactively. Although originally designed for helping users choose routines from NAG or other libraries, it has been used in different areas as well. There is a NITPACK package to assist a user in choosing a routine to solve a boundary value ordinary differential equation from the NAG, Harwell, and IMSL Libraries. In addition, systems like NAG's online help package and GAMS [Boisvert *et al.* 1985] will provide a certain amount of assistance to the user.

A more up-to-date package which exploits modern interface technology (mice, hypertext, X-windows etc.) is currently being developed by NAG. The KASTLE system [NAG LTD. 1989], does not use the interrogative "yes / no" style interface of the previous systems. Rather, users are presented with a list of *attributes* which may define the problem. At any time they may ask for more details about an attribute. The user selects attributes which consequently narrow the search space and so the display is updated. When the user is finally satisfied that the problem to be solved is completely described, he or she may ask the program to recommend a list of routines in order of preference.

The fundamental drawback with all these systems is that the user still has to decide whether various properties hold for his or her particular problem. In some ways this is not an entirely unreasonable approach. Deciding whether or not the expression

$$e^{-\frac{1}{x}}$$

has a singularity at the point $x = 0$ is not easy to do automatically; but on the other hand deciding whether a matrix is symmetric is tedious but trivial. Clearly it would

```
6: integrate(integrand(x)=(1-log(x))^(-10),region=[0:1]);

{ALIST,BLIST,ELIST,RLIST,ABSERR,INTEGRAL,INTERVALS}

7: integral;

0.0989291326406463
```

Figure 11-1: An example of the use of the automatic routine chooser.

be highly desirable if a system existed which could examine a user's problem and give advice on the best method to solve it.

This is clearly an area where a computer algebra system could be very useful, since we wish to do a symbolic analysis of the problem. With IRENA, however, we can go one step further and actually make the call to the NAG Library automatically.

11.1 ARC — An Automatic Routine Chooser.

We have implemented a system in Reduce which, given a user's description of the problem to be solved, chooses the best routines to use according to a given set of criteria. If there are several candidate routines then they are ranked according to which seems the most promising. Calls to IRENA are then made until either a result is obtained or the candidate list is exhausted. So far only a subset of the D01 (quadrature) chapter has been implemented, though the system should be extensible to other suitable areas of the Library.

Consider the example shown in figure 11-1. The first thing to note is that the parameters the user has provided are the same as in our jazzed interface to the D01 chapter. This is hardly surprising since we set out to provide a canonical interface to each chapter, which was a complete representation of the problem being solved. No indication is given of which routine has been used to solve the problem (in fact it was D01AKF; failing that ARC also suggested using D01AJF), or what criteria have been used to determine this. To get more detailed information, the user should set the switch

selectinfo in which case the system will provide details about its progress, as shown in figure 11-2. More examples can be found in appendix F.

11.1.1 The Basic Strategy.

ARC starts by processing the arguments provided by the user. The initial set of candidate routines is then chosen on the basis of the *apparent* structure of these arguments. For example in the case of integration we have three sets of routines dealing with three distinct areas — one dimensional finite quadrature, one dimensional semi-infinite or infinite quadrature, and multi-dimensional finite quadrature — and can determine which case we have by a fairly superficial examination of the parameters provided by the user[1].

We now have a list of routines, associated with each of which is a list of predicates. These predicates are of two kinds: those whose satisfaction is necessary for a routine to be chosen; and those which, if satisfied, should remove that routine from consideration. We loop round, choosing a predicate to evaluate each time. Currently we choose the predicate which is associated with the most routines. This is a reasonable strategy since there is not a great deal of difference in the cost of evaluating predicates. Should we encounter a situation elsewhere in the Library where this is not the case we would probably adopt a different strategy. At each stage some routines are *matched*, i.e. they are satisfied by the value of the predicate; while others are *pruned*. Thus as we move through the predicates we have a steadily shrinking list of routines, each of which has a steadily shrinking set of predicates associated with it. Eventually we are left with a set of routines, all of whose predicates have been matched. In theory this set could be empty but, due to the structure of the current knowledge base, we are guaranteed that at least one routine will be recommended[2].

The routine list is then ordered. Currently we use a simple ordering based on the initial number of predicates associated with each routine: the idea being that the greater this number then the more specialised the routine, and the more likely it is that it will

[1]There is also the case where we have a one-dimensional integral whose functional form is not known, but since there is only one routine in the library to deal with this case we have ignored it.

[2]In fact, at least one of the routines D01AJF, D01ATF, D01AHF will always be recommended.

```
9: integrate(integrand(x)=(1-log(x))^(-10),region=[0:1]);
* VECTOR-PROCESSOR is NIL
--> The following routines are being pruned: (D01ATF D01AUF)
--> The following routines have been matched: (D01AJF D01AKF)
* SMOOTH-EXCEPT-W2 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01ANF D01AQF D01ALF)
* CONTINUOUS is T
--> The following routines are being pruned: (D01AQF)
--> The following routines have been matched: (D01AKF)
* KNOWN-SINGULARITIES is NIL
--> The following routines are being pruned: (D01ALF)
--> The following routines have been matched: (D01AKF D01ANF)
* CONTINUOUS-EXCEPT-AT-END-POINTS is NIL
--> The following routines are being pruned: (D01AHF)
--> The following routines have been matched: (D01AJF D01AKF D01ANF
D01BDF)
* REASONABLY-SMOOTH-EXCEPT-WEIGHT is NIL
--> The following routines are being pruned: (D01ANF)
--> The following routines have been matched: (D01AKF D01APF D01BDF)
* CONTINUOUS-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01APF)
--> The following routines have been matched: (D01AKF D01BDF)
* CONTINUOUS-EXCEPT-W2 is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01AKF D01BDF)
* REASONABLY-SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BDF)
--> The following routines have been matched: NIL
*** The recommended routines are :
      (D01AKF D01AJF)
* Now calling routine D01AKF
{ALIST,BLIST,ELIST,RLIST,ABSERR,INTEGRAL,INTERVALS}
```

Figure 11-2: ARC's tracing facility.

give a result. This is true due to the number of predicates which, if satisfied, will cause the routine to be pruned: for the more general routines the outcome of evaluating many predicates is immaterial. While obviously depending on the structure of our knowledge base, this seems to work well in practice.

Finally we make a call to our first choice of routine, using IRENA. If on exit we encounter a non-zero ifail, or an IEEE exception occurs, the next routine in the list is tried and so on. If we get a result then this is returned to the user, otherwise a message is printed listing the routines which we have tried.

11.1.2 The D01 Knowledge Base.

Much of the success of the relatively simple strategy outlined in § 11.1.1 is dependent on the structure of the knowledge base. In this section we shall look at the various predicates, and how they relate to the various routines.

The current knowledge base deals only with one-dimensional finite quadrature. It contains sixteen entries for twelve routines, in other words some routines (D01ARF and D01BAF) can be used in different circumstances and so have multiple entries. There are a total of eleven predicates used to distinguish between them:

- **Continuous** The integrand is continuous over the region.
- **Known-singularities** The integrand has singularities within the region, but we know where they all are.
- **Continuous-except-at-end-points** The only singularities of the integral in the region lie at one or both of its end points.
- **Smooth** Both the integrand and its derivative are continuous, and the integrand does not oscillate very much.
- **Reasonably-smooth** Both the integrand and its derivative are continuous.
- **Vector-processor** The user has indicated that the target architecture is a vector processor, by providing the keyword *vector-processor* in the key list.

- **Continuous-except-w1** The integrand is continuous over the region apart from the weight function:

$$(b-x)^{\alpha} * (x-a)^{\beta} * (\log(b-x))^{k} * (\log(x-a))^{l}$$

where a and b are the end points of the region of integration and x is the independent variable.

- **Continuous-except-w2** The integrand is continuous over the region apart from the weight function:

$$\frac{1}{(x-c)}$$

where c is neither of the endpoints of the region and x is the independent variable.

- **Smooth-except-w1** The integrand is *smooth* (see above) apart from the weight function:

$$\left|x - \frac{a+b}{2}\right|^{c}$$

where a and b are the end points of the region of integration and x is the independent variable.

- **Smooth-except-w2** The integrand is *smooth* (see above) apart from the weight function:

$$(b-x)^{c}(x-a)^{d}$$

where a and b are the end points of the region of integration and x is the independent variable.

- **Reasonably-smooth-except-weight** The integrand is *reasonably smooth* (see above) apart from one of the weight functions:

$$\cos(\omega x) \qquad \sin(\omega x)$$

where x is the independent variable.

These predicates have been determined from the decision tree in the NAG Manual.

What they actually mean (e.g. in the case of *reasonably-smooth*) has sometimes been a matter of guesswork and experimentation.

Obviously some of these predicates are inter-related. For example if we know where all the singularities are we can tell if they all lie at the end points of the region of integration. However it is less clear whether for example the integrand being continuous should preclude the predicate *continuous-except-w1* from holding since if both α and β are positive integers there are in fact no singularities in the weight. This possibly misleading choice of name comes from the decision tree in the NAG manual. Which predicates are associated with which routines are given in Appendix E.

11.2 Implementation details.

In this section we shall describe how the predicates are implemented, and how the link to IRENA is made.

11.2.1 The Predicates.

The efficiency of the system is largely determined by the efficiency of evaluating the predicates. Put bluntly, there is no point spending an hour locating the singularities of the integrand when you could have tried each and every relevant routine in that time. In fact, the feedback from such an exercise would probably help locate the singularities far more quickly. Thus in some cases we have opted for a "quick but dirty" approach rather than a slow but thorough one.

Testing continuity.

Our first experiments were with methods involving Padé approximations and asymptotic expansions of Taylor series, but these turned out to be far too slow for our purposes. We thus adopted a less adventurous strategy which would locate poles and some logarithmic singularities while indicating where there was a possibility that some singularities existed but had not been detected.

We start with a quotient of polynomials (in the Reduce sense: i.e. $\sin(x)$ is a perfectly

valid polynomial). Poles of this expression correspond to zeros of the denominator. So we factorise the denominator and find the zeros of the factors using the Reduce roots package [Kameny 1990] if the factor is a polynomial (in the mathematical sense), and the Reduce solve package otherwise. In the former case we can specify that the roots must lie in a particular region, but in the latter we have to check which ones do. This is complicated by the fact that the answer may involve arbitrary integers (e.g. the roots of $\sin(x) = 0$ are $x = n\pi$ where n is any integer). If solve fails to find any roots, or we are unable to decide whether a root lies in the region of integration, we signal that some singularities *may* exist which we have been unable to detect.

This approach has the merit of being fast and in most cases will, if anything, err on the conservative side. The one exception to this is when we have a removable singularity: e.g. if the initial expression is

$$f(x) = \frac{\sin(x)}{x}$$

then our approach will suggest a singularity at $x = 0$ even though $f(0) = 1$. In actual fact this is not necessarily a bad thing, since a FORTRAN function would probably evaluate this incorrectly. Integrating $f(x)$ between -1 and 1 using D01AJF yields an IEEE exception, but using D01ALF and telling it that a singularity exists at the origin yields the correct result.

The other kind of singularities which we look for are logarithmic singularities in the numerator. This is done by looking at each factor of the numerator, identifying logarithm terms, and checking whether their arguments equal zero anywhere in the region of integration. For this we use the same techniques as when looking for poles. A final check is made to ensure that none of the logarithmic singularities we have found actually disappear because another factor of the integrand is zero at that point (i.e. we want to avoid claiming that e.g. $x\log(x)$ has a singularity at $x = 0$).

Estimating oscillations.

One method of calculating the number of times a function oscillates in an interval is to count the number of zeros of its derivative. We have adopted a method for estimating this as follows:

1. Randomly choose a set of sample points in the interval.

2. Evaluate the function at each point.

3. Count up the number of sign changes and zeros.

This gives us a lower bound on the number of zeros. We can then compare this with some critical value to decide whether the function oscillates "too much". Doing lots of function evaluations in Reduce's algebraic mode is rather slow, so the current implementation of this package does everything in symbolic mode which speeds the process up by an order of magnitude, and so allows us to examine more points at the cost of a slight restriction on the sorts of functions which we can handle. Taking a smaller grid of points would probably require an adaptive strategy where, if a small number of zeros were found, we examined their neighbourhood to see if there were any more. The current grid size is fifty points and the critical number of zeros is ten, which is probably rather conservative.

Identifying Weights.

For this we use the PM pattern matcher [McIsaac 1990] which, given an expression template and an expression, will try to match the unknowns in the template to values in the expression. For example in:

```
38:  m(20*sin(6*x+pi),?a*sin(?b*?v+?c));

{?A->20,?B->6,?C->PI,?V->X}
```

we are matching the unknowns in the template on the right with the actual expression on the left. Unfortunately there are two drawbacks with using this package. The first is that PM doesn't know anything about mathematics and so will not match e.g. the template `?a*x` with the expression x where $a \rightarrow 1$. This sort of problem can be circumvented by defining families of templates rather than just one, incorporating all the various possibilities. The second problem is that Reduce will sometimes re-order an expression in such a way as to fool the pattern matcher. For example the expression

$(1-x)^3$ will not match the template `(?a-x)^n` because Reduce rewrites it as $-(x-1)^3$ (this depends on the setting of various switches). Thus writing templates is something of an art form.

Another problem with PM is that it is incompatible with some other Reduce packages because it redefines several fundamental operators to suit itself. We have had to adapt it to allow the Roots package to run correctly after PM has been loaded.

11.2.2 The link to IRENA.

Provided that the switch *irenalink* has been set, ARC will attempt to call the selected routines using IRENA.

For most routines this is simply a case of passing the user's parameters to the chosen routine via IRENA. This is a consequence of our consistent jazzing of interfaces to routines in the same chapter. In some cases a little extra work is required: for example where we also need to provide a vector of singularities; or where we have to provide the parameters in the weight function separately. Routines in this category have a *setup* function to perform these operations.

We have suppressed the printing of `IFAIL` and IEEE messages for failed routines to improve the ARC interface to the user. One other improvement which would be nice would be to change some of the output names of the parameters to reflect the different way in which the routines have been called. For example, D01ANF computes the sine or cosine transform of a function g, i.e.

$$\int g(x)\sin(\omega x)\,dx$$

or

$$\int g(x)\cos(\omega x)\,dx$$

The result is called (via the jazz system) *transform* which makes sense if the routine is called directly, but when called via ARC the name *integral* would be better.

11.3 Future Developments.

Obviously we would like to extend ARC to other suitable areas of the Library. Some chapters would be trivial, for example in the C02 chapter which contains routines for finding the zeros of polynomials we need only decide whether the given expression is a quadratic or not, and whether the coefficients are complex. Some chapters are inappropriate for this approach, for example chapter F06 consists of linear algebra support routines — i.e. operations on matrices and vectors — where each routine essentially performs a different task.

There are other improvements that could be made. It might, for example, be worthwhile using the `IFAIL` value from an unsuccessful call to influence the subsequent choice of routine. The disadvantage with this is that we would need much more information about each routine in the database. A variation on this might be to make an initial attempt at an integral using the most general routine, D01AJF, and if this failed then use the value of the `IFAIL` diagnostic and the partial results which the routine returned to decide how to proceed. While this is an interesting idea, it has the drawback that the diagnostics are not particularly distinct (three of the five non-trivial ones suggest that a singularity or discontinuity has been detected which cannot be handled) or comprehensive. In addition many areas of the Library do not have a "most general" routine, so the idea does not extend to other areas.

Chapter 12

Conclusions

There have been two main aims in the work discussed in the previous chapters. The first has been to provide an environment in which both algebraic and numerical algorithms can be applied profitably and effectively for the solution of mathematical problems. The second has been to make the extensive high-quality methods available in numerical subroutine libraries available in an easy-to-use form to todays modern breed of computer user. Both these aims are satisfied by IRENA; though ARC provides an even easier, though currently incomplete, interface. Throughout the design of these systems we have tried to be *non-authoritarian*, that is to say we have never *forced* a user to make a decision, though we have given as much sensible advice as we can. If the user knows what value he or she wants for a control parameter then he or she can provide it, otherwise the system default will be chosen. If the user knows which routine is the best to solve a given problem then it can be called directly, otherwise ARC will try to select one. Where IRENA is a simplified interface to the NAG Library, ARC is in some sense a simplified interface to IRENA.

12.1 Side-effects of developing IRENA.

There have been other areas where this work has made an impact. Based on the kind of information we required to build our IRENA interfaces, NAG have redesigned their documentation to facilitate the future design of other packages which call Library

routines; and of knowledge-based, advice-giving systems. The design of the interfaces themselves might also be useful, for example in designing a suite of FORTRAN-90 [ANSI 1989] interfaces to the current Library. In addition the adoption of canonical interfaces to areas of the Library could be used as guidelines by future designers of routines.

Not only have we identified what a package designer wants from a subroutine library, but we now have a better idea of what facilities a package designer needs the host computer algebra system to provide. In some ways the main use that IRENA makes of Reduce itself is as a user interface and parser, so it would be nice to have tools to make the process of building parsers in LISP easier. The main area however where we have clearly seen the need for extra functionality is in the code generation and dynamic linking processes. In IRENA some of the new features have found their way into GENTRAN, others have been solved by a suite of special functions which, with hindsight, should perhaps be collected together and made available separately from IRENA to other users. We now feel that any system to support major package development should, as a bare minimum, support the following in addition to the simple ability to translate expressions:

- **Skeletal Program Facilities** GENTRAN's template facilities are quite good, and they are particularly amenable to automatic generation. Unfortunately they require the designer to know a fair amount about the target language. A more abstract description of program segments — perhaps a more general package like our ASP descriptor language described in chapter 6 — would also be extremely useful.

- **Cross Architecture Support** Essentially the user should be able to determine the precision of the generated code, including the number of digits printed for floating point numbers and, in FORTRAN, the names of intrinsic functions.

- **Access to the Parser** It is extremely useful to be able to tell if certain tokens have appeared in the input, and to specify "special" treatment for them. In IRENA this is used to determine whether irena constants (see § 3.6.1) have been

encountered, but one can envisage other uses for it as well. For example, the names of functions might change according to the chosen precision, machine type etc. In IRENA we use a piece of dedicated code within GENTRAN to spot and transform certain tokens, but a more general mechanism ought to be made available.

- **Optimisation** It is essential that there are facilities for automatic optimisation of sequences of expressions, not necessarily provided together at the one time.

- **Symbol Table** The user should be able to maintain a symbol table of type and array declarations etc. The current GENTRAN symbol table is not powerful enough to support properly block-structured languages, pointers, or user-defined data structures. Since C is rapidly overtaking FORTRAN in popularity among scientific users, and FORTRAN-90 supports many of these facilities, any new package should take account of them.

- **Dynamic Linking** A user should be able to call pieces of foreign code without the need to statically link them into the host algebra system. How IRENA does this is described in chapter 7, but it is not a very general system. An alternative approach might be to use remote procedure calls to shared libraries. A suite of functions to convert the generated FORTRAN objects into algebraic objects is also necessary.

12.2 Further Work.

In many ways the work described in this thesis is a beginning, rather than an end in itself. We have provided package developers with a high-quality suite of tools which they can apply to specific problem domains. These packages may be concerned with solving a particular problem (e.g. determining rate laws in biochemistry), or a particular class of problems (e.g. solving definite integrals or differential equations). The latter are interesting since they could exploit both symbolic and numerical techniques.

Integration is particularly interesting, since there exist sophisticated algebraic "black-box" integrators which can solve a wide range of problems. In many of these

cases the algebraic integrator will yield a result faster than IRENA will, but sometimes it will be extremely slow or will run out of memory. Numerical methods have the advantage that, because in general they have a fixed limit on the number of iterations they perform, they will terminate in a predictable time (in some cases they can be continued if a result has not been found, as in the example in § 10.2). There are thus three questions which it might be worth asking:

1. Can the algebraic integrator solve this integral?
2. Will it solve it in "reasonable" time?
3. Does it have the resources (e.g. memory) to solve it?

While it seems possible to determine a reasonable answer to the first question, the others appear rather more difficult.

Another interesting problem arises when the algebraic integrator can only yield a partial solution. The question then arises, is it better to perform the whole integration numerically, or to attempt only the residue left? [Davenport 1985] gives as an example of a case where analytic integration does indeed make numerical integration easier the reduction:

$$\int_0^y \log \sin x \, dx = y \log \sin y - \int_0^y x \frac{\cos x}{\sin x} \, dx$$

In this case the reduction has removed the log term from the integral, and indeed made the problem more tractable. However it is not always the case that analytic integration makes a problem easier. The above reduction was done in Scratchpad, an example with the Reduce integrator which causes problems is:

$$\int_0^1 e^x \log x \, dx = e^x \log x \; |_0^1 - \int_0^1 \frac{e^x}{x} \, dx$$

While in both these cases it is clear which is the better form, this is not always going to be the case. Indeed, while the general–purpose NAG integration routine D01AJF handles both these examples in their original form with ease, it can manage the first problem's reduction, but not the second (see figure 12.2).

```
2: d01ajf(integrand(x)=log(sin x),region=[0:1]);

{ALIST,BLIST,ELIST,RLIST,ABSERR,INTEGRAL,INTERVALS}

3: integral;

 - 1.0567202059916

4: abserr;

1.1102230246252E-15

5: d01ajf(integrand(x)=x*cos(x)/sin(x),region=[0:1]);

{ALIST,BLIST,ELIST,RLIST,ABSERR,INTEGRAL,INTERVALS}

6: on numval;

7: log(sin 1) - integral;

 - 1.0567202059916

8: d01ajf(integrand(x)=e^x*(log x),region=[0:1]);

{ALIST,BLIST,ELIST,RLIST,ABSERR,INTEGRAL,INTERVALS}

9: integral;

 - 1.3179021514544

10: off print!-ifail!-message; % Inhibit verbose error message

11:  d01ajf(integrand(x)=e^x/x,region=[0:1]);
 ** The maximum number of subdivisions (LIMIT) has been reached:
    LIMIT =              500   LW =        2000   LIW =       500
 ** ABNORMAL EXIT from NAG Library routine D01AJF: IFAIL =      1
 ** NAG soft failure - control returned

{ALIST,BLIST,ELIST,RLIST,ABSERR,INTEGRAL,INTERVALS}
```

Figure 12-1: Evaluating partially processed integrals.

Another area which we have not tackled arises when the solution of a problem requires the "coupled" use of several NAG routines. For example consider the problem

$$\min_{a \le x \le b} \int_{\alpha}^{\beta} f(x,t)\, dt$$

where the integral must be calculated numerically. The structure of the program will be a little messy: the optimisation routine will take an ASP which will call an integration routine. The value of x will need to be kept in a `COMMON` block because of the fixed form of the various ASPs involved. However the real difficulties arise because of the way in which the choice of parameters in one routine effect the values which should be chosen in another. The NAG manual normally recommends that the tolerance in an optimisation routine is the square root of the machine precision: approximately 1e-08 on a SUN workstation. However it is usual for integration routines to be called with a lower tolerance than this: in IRENA the usual tolerance is 1e-04. It is clearly nonsense to try and use an accuracy of 1e-08 to distinguish data accurate only to 1e-04. Thus for this kind of application we would need an "intelligent" defaults system which understood the relationships between parameters in different routines.

12.3 Summary.

The systems we have built open up a lot of possibilities. We now have a toolkit of techniques which can be used to solve specific problems. It is to be hoped that in future they will be exploited to the full by package developers, as well as by general users. The techniques we have described are not limited to Reduce and the NAG Library: other Libraries could be interfaced to Reduce using these techniques; and the interfaces to the NAG routines could be re-used in a package based around a different computer algebra system.

The fundamental aim of this work is very simple. We believe that people with mathematical problems want mathematical solutions and, provided that they can be relied upon, aren't generally particularly interested in where they came from. We believe the work described here is a major step towards that goal.

Appendix A

Notation for syntax figures.

A.1 Conventions.

Terminal symbols appear in **bold** typeface.

Non-terminal symbols are surrounded by a pair of angled brackets thus: < ... >.

A.2 Symbols.

::=	assigns a definition to a non-terminal symbol
||	or
{ ... }	an optional part
(...)	a group
*	repeated zero or more times
+	repeated one or more times
+n	repeated n times

N.B. Notice the difference between e.g. **{** and {.

Appendix B

Changes to the REDUCE and GENTRAN systems made for IRENA.

B.1 Introduction

IRENAreduce is the standard reduce plus the following extra bits of standard PSL:

- oload.b
- defmacro.b
- defmacro1.b
- defmacro2.b
- l2cdatacon.b
- s-strings.b
- entry.b
- matrix.b

The latter is necessary because IRENA builds matrices without using any of the explicit operators which would load *matrix.b* if it wasn't there. To do this it is necessary to load *entry.b* first. In addition we load:

- irena_oload.b
- irenamisc.b
- cvec2.b

which have been specially written. The first is described in § 7.3 and the second is used to catch IEEE exceptions during the execution of foreign code. The third contains functions for accessing different types of bitmaps in memory, i.e. for reading the results left on the heap by the FORTRAN program. In addition some extra procedures for manipulating strings are added. Finally the GENTRAN package and the code optimiser are both pre-loaded.

The rest of this document lists the changes to GENTRAN, and PSL made by the IRENA project at Bath. It doesn't mention bug fixes, only added features.

B.2 New public GENTRAN features

The following additional features have been added to the GENTRAN system. They are all of use to general users.

B.2.1 DOUBLE

With this switch ON, the following happens:

- Declarations of parameters of appropriate type are converted to their double precision counterparts. In Fortran and Ratfor this means that objects of type `REAL` are converted to objects of type `DOUBLE PRECISION` and objects of type `COMPLEX` are converted to `COMPLEX*16` [1]. In C the counterpart of `float` is `double`, and

[1] This is not part of the ANSI Fortran standard. Some compilers accept `DOUBLE COMPLEX` as well as, or instead of, `COMPLEX*16`, and some accept neither.

of int is long. There is no complex data type and trying to translate complex objects causes an error.

- Similarly subprograms are given their correct type where appropriate.
- In Fortran and Ratfor REAL and COMPLEX numbers are printed with the correct double precision format.
- Intrinsic functions are converted to their double precision counterparts (e.g. in Fortran SIN → DSIN etc.).

B.2.2 Intrinsic Functions

A warning is issued if a standard REDUCE function is encountered which does not have an intrinsic counterpart in the target language (e.g. *cot*, *sec* etc.). Output is not halted in case this is a user–supplied function, either via a REDUCE definition or within a GENTRAN template.

Where the arguments of intrinsic functions are of the incorrect type they are converted to the correct one. Where this cannot be done in advance, i.e. they are variables whose types have not been declared using the *declare* function, the correct coercion function is generated. In other words *sin(x)* will be converted to `SIN(REAL(X))` or `DSIN(DBLE(X))`, depending on the setting of *DOUBLE*.

B.2.3 Complex Numbers

With the switch *COMPLEX* set ON, GENTRAN generates the correct representation for a complex number in the given precision provided that:

1. The current language supports a complex data type (if it doesn't then an error results);
2. The complex quantity is evaluated by REDUCE to give an object of the correct domain, i.e.

```
gentran x:=: 1+i;
```

```
gentran x:= eval 1+i;


z := 1+i;
gentran x:=: z;
```

will all generate the correct result, as will their Symbolic mode equivalents, while:

```
gentran x := 1+i;
```

will not.

B.2.4 GETDECS

With this switch ON, the following happens:

1. The indices of loops are automatically declared to be integers.

2. There is a global variable *deftype**, which is the default type given to objects. Subprograms, their parameters, and local scalar objects are automatically assigned this type (except in the cases below). Note that types such as *real*8* or *double precision* should not be used as, if *DOUBLE* is on, then a default type of *real* will in fact be *double precision* anyway.

3. Standard Gentran does not accept *scalar* (or *real*, *integer*) local declarations, or subprogram declarations of the form:

   ```
   INTEGER PROCEDURE ...
   REAL PROCEDURE ...
   ```

   Enhanced Gentran however does. Local scalars are assigned the value of *deftype**, and locally declared *integer*s and *real*s are declared accordingly (though with consideration of precision if the flag *DOUBLE* is on). If a procedure declaration is preceded by either *INTEGER* or *REAL* then not only that subprogram but also its parameters are assigned that type (again with reference to the current precision).

B.2.5 Types

A check is made on output to ensure that all types generated are legal ones. This is necessary since *deftype** can be set to anything. Note that *deftype** ought normally to be given a simple REDUCE type as its value, such as *real*, *integer*, or *complex*, since this will always be translated into the corresponding type in the target language on output.

B.2.6 Modified PERIOD flag

If the *PERIOD* flag is on then, if you have already declared something to be an *integer*, when you come to assign a value to it that value will not automatically be coerced to a *real*. This works whatever method has been used to set the type (i.e. if *GETDECS* is on this works for integers which have been declared automatically).

B.2.7 KEEPDECS

In the Bath version of Gentran an entry is removed from the symbol table once a declaration has been generated for it. The *KEEPDECS* switch (by default OFF) disables this, allowing a user to check the types of objects which GENTRAN has generated (useful if they are being generated automatically).

B.2.8 MAKECALLS

A statement like:

```
gentran x^2+1$
```

will yield the result:

```
X**2+1
```

but, under normal circumstances, a statement like:

```
gentran sin(x)$
```

will yield the result:

```
      CALL DSIN(X)
```

The switch *MAKECALLS* (OFF by default) will make GENTRAN yield

```
DSIN(X)
```

This is useful if you do not know in advance what the form of the expression which you are translating is going to be.

B.2.9 E

When GENTRAN encounters e it translates it into EXP(1), and when GENTRAN encounters e^x it is translated to EXP(X). This is then translated into the correct statement in the given language and precision. Note that it is still possible to do something like:

```
gentran e:=:e;
```

and get the correct result.

B.3 Private Gentran Features

Some code has been added to procedure *lispcodeexp* to spot the occurrence of certain variable names such as *PI* for IRENA.

B.4 Additions to the Code Optimiser

These are changes to the code optimiser written by Barbara Gates.

B.4.1 Domain elements

The optimiser does not attempt to manipulate domain elements, e.g. to decide whether one divides another.

B.4.2 DECLARECSENAMES

With this switch ON the new variables generated for the common sub-expressions found by the optimiser are placed in the GENTRAN symbol table at the current position and declared to be of type *tempvartype**. It is ON by default.

B.4.3 OPTIMISEWAIT

This feature is essential for using the optimiser with IRENA. When the switch *OPTIMISEWAIT* is set ON, GENTRAN does not optimise or evaluate any expressions it is given, but instead stacks them. When the user issues a call to the function *Optimise-Stack* all the stacked expressions are passed to the optimiser and the results are then processed by GENTRAN and returned. A typical piece of code which uses this is:

```
on optimisewait;
for i:=1:n do sym!-gentran '(lrsetq (f i) (getval mkid 'f i));
optimise!-stack();
```

Here we have a series of expressions f_i which we wish to evaluate and assign to the elements of the matrix f. The effect of this is the same as if we had gone:

```
gentran <<
        f(1) :=: getval 'f1;
        f(2) :=: getval 'f2;
          :   :    :      :
        f(n) :=: getval 'fn;
        >>;
```

though of course in this case we would need to know the value of n in advance.

Appendix C

Matrix Representation in IRENA.

All IRENA matrices are represented as lists of lists. In most cases, the inner lists represent all rows (or partial rows) in the natural order. For strict upper (lower) triangular matrices, the last (first) inner list is empty, although this empty list may optionally be omitted, where there is no possibility of confusion between strict upper and lower triangular matrices (i.e. for strict triangular matrices whose order is more than 2). In general we do not differentiate between upper and lower forms, where the form can be detected automatically. A full list of IRENA matrix types appears in tables C.1–C.3. There is, additionally, an IRENA vector type, represented as a single list.

Type	Representation
band (fixed bandwidth)	each inner list specifies a "diagonal"
symmetric band (fixed bandwidth)	only the superdiagonal and diagonal (or diagonal and subdiagonal) lists

Table C.1: Matrices with diagonal lists: uppermost diagonal first throughout

Type	Representation
full	each inner list specifies a row
symmetric	each inner list specifies that part of a row for which $i \geq j$ or $i \leq j$
skew-symmetric	each inner list specifies that part of a row for which $i \geq j$ or $i \leq j$
Hermitian	each inner list specifies that part of a row for which $i \geq j$ or $i \leq j$
strict upper triangular	each inner list specifies that part of a row for which $i < j$
upper triangular	each inner list specifies that part of a row for which $i \leq j$
upper Hessenberg	each inner list specifies that part of a row for which $i \leq j + 1$
strict lower triangular	each inner list specifies that part of a row for which $i > j$
lower triangular	each inner list specifies that part of a row for which $i \geq j$
lower Hessenberg	each inner list specifies that part of a row for which $i \geq j - 1$
general band (variable bandwidth)	each inner list specifies that part of a row lying within the envelope, the list being packed out with zeroes for symmetry about the diagonal
symmetric band (variable bandwidth)	each inner list specifies that part of a row, lying within the envelope, for which $i \geq j$
(variable bandwidth)	within the envelope, for which $i \geq j$

Table C.2: Matrices with row lists: uppermost row first throughout (i represents the row, and j the column index).

Type	Representation
sparse	3 inner lists, each in same arbitrary order, containing: first list — row indices of non-zero elements second list — column indices of non-zero elements third list — non-zero elements
symmetric sparse	as sparse, restricted to either upper or lower triangle.

An additional representation of sparse matrices is allowed, since, in some circumstances, entering these would be less error prone:

long sparse	a list of triples {r,c,v} representing the row index, column index and value, respectively, of the non-zero elements (in arbitrary order)
symmetric long sparse	as long sparse, restricted to either upper or lower triangle

Table C.3: Sparse matrices

Appendix D

IRENA CONSTANTS

There are a number of routines in the NAG Library which determine various system dependent quantities. They can be used in IRENA (either interactively while values are being given, or in the defaults files) through simple mnemonics. As explained in § 3.6.1 this causes proper calls and assignments to be generated in the FORTRAN. The following is a complete list of those in the current system, routines which have been superseded but retained in the Library are not included.

Name	NAG Routine	Purpose
pi	X01AAF	The value of π.
fpbase	X02BHF	The base of the computer's arithmetic.
fpprec	X02BJF	The precision.
fpemin	X02BKF	The minimum exponent in floating point numbers.
fpemax	X02BLF	The maximum exponent in floating point numbers.
fprnds	X02DJF	Returns `.TRUE.` if rounding is always correct in the final bit, `.FALSE.` otherwise.
fpeps	X02AJF	The smallest number ϵ such that $1 + \epsilon > 0$.
fptiny	X02AKF	The smallest positive floating point number.
fphuge	X02ALF	The largest positive floating point number.
fprnge	X02AMF	The smallest positive floating point number z such that, for any x in $[z, 1/z]$, the following may be "safely" calculated: $-x$, $\frac{1}{x}$, $\sqrt{x}$, $\log(x)$, $\exp(\log(x))$, $y^{\frac{\log(x)}{\log(y)}}$ $\forall y$
scmaxa	X02AHF	The largest number for which `SIN` and `COS` return a result with some meaningful accuracy.
maxint	X02BBF	The largest positive integer.
fpdigs	X02BEF	The number of decimal digits which can be relied upon in floating point numbers.
asetsz	X02CAF	The estimated active set size in a paged environment, otherwise zero.
ufevnt	X02DAF	Returns `.FALSE.` if underflowing numbers are set to zero, otherwise `.TRUE.`.
defner	X04AAF	The FORTRAN unit number for advisory messages.
defnad	X04ABF	The FORTRAN unit number for error messages.

Appendix E

The ARC Predicates

This appendix lists the current predicates for the various routines in the database. As can be seen, two of the routines (D01BAF and D01ARF) appear several times to handle several situations.

D01ARF

T-PREDICATES: *SMOOTH-INTEGRAND*

F-PREDICATES: *CONTINUOUS-EXCEPT-W1*

CONTINUOUS-EXCEPT-W2

REASONABLY-SMOOTH-EXCEPT-WEIGHT

CONTINUOUS-EXCEPT-AT-END-POINTS

D01ARF

T-PREDICATES: *SMOOTH-EXCEPT-W1*

F-PREDICATES: *CONTINUOUS-EXCEPT-W1*

CONTINUOUS-EXCEPT-W2

REASONABLY-SMOOTH-EXCEPT-WEIGHT

CONTINUOUS-EXCEPT-AT-END-POINTS

D01ARF

T-PREDICATES: *SMOOTH-EXCEPT-W2*

F-PREDICATES: *CONTINUOUS-EXCEPT-W1*

CONTINUOUS-EXCEPT-W2

REASONABLY-SMOOTH-EXCEPT-WEIGHT

CONTINUOUS-EXCEPT-AT-END-POINTS

D01BAF

T-PREDICATES: *SMOOTH-INTEGRAND*

F-PREDICATES: *CONTINUOUS-EXCEPT-W1*

CONTINUOUS-EXCEPT-W2

REASONABLY-SMOOTH-EXCEPT-WEIGHT

CONTINUOUS-EXCEPT-AT-END-POINTS

D01BAF

T-PREDICATES: *SMOOTH-EXCEPT-W1*

F-PREDICATES: *CONTINUOUS-EXCEPT-W1*

CONTINUOUS-EXCEPT-W2

REASONABLY-SMOOTH-EXCEPT-WEIGHT

CONTINUOUS-EXCEPT-AT-END-POINTS

D01BAF

T-PREDICATES: *SMOOTH-EXCEPT-W2*

F-PREDICATES: *CONTINUOUS-EXCEPT-W1*

CONTINUOUS-EXCEPT-W2

REASONABLY-SMOOTH-EXCEPT-WEIGHT

CONTINUOUS-EXCEPT-AT-END-POINTS

D01BDF

T-PREDICATES: *REASONABLY-SMOOTH-INTEGRAND*

F-PREDICATES: *CONTINUOUS-EXCEPT-W1*

CONTINUOUS-EXCEPT-W2

REASONABLY-SMOOTH-EXCEPT-WEIGHT

CONTINUOUS-EXCEPT-AT-END-POINTS

D01ALF

T-PREDICATES: *KNOWN-SINGULARITIES*

F-PREDICATES: *SMOOTH-INTEGRAND*

SMOOTH-EXCEPT-W1

SMOOTH-EXCEPT-W2

REASONABLY-SMOOTH-INTEGRAND

REASONABLY-SMOOTH-EXCEPT-WEIGHT

D01APF

T-PREDICATES: *CONTINUOUS-EXCEPT-W1*

F-PREDICATES: *CONTINUOUS-EXCEPT-W2*

REASONABLY-SMOOTH-EXCEPT-WEIGHT

D01AQF

T-PREDICATES: *CONTINUOUS-EXCEPT-W2*

F-PREDICATES: *SMOOTH-INTEGRAND*

SMOOTH-EXCEPT-W1

SMOOTH-EXCEPT-W2

REASONABLY-SMOOTH-INTEGRAND

CONTINUOUS

CONTINUOUS-EXCEPT-W1

REASONABLY-SMOOTH-EXCEPT-WEIGHT

CONTINUOUS-EXCEPT-AT-END-POINTS

D01ANF

T-PREDICATES: *REASONABLY-SMOOTH-EXCEPT-WEIGHT*

F-PREDICATES: *SMOOTH-INTEGRAND*

SMOOTH-EXCEPT-W1

SMOOTH-EXCEPT-W2

CONTINUOUS-EXCEPT-W1

CONTINUOUS-EXCEPT-W2

CONTINUOUS-EXCEPT-AT-END-POINTS

KNOWN-SINGULARITIES

D01AKF

T-PREDICATES: *CONTINUOUS*

F-PREDICATES: *CONTINUOUS-EXCEPT-W1*

CONTINUOUS-EXCEPT-W2

SMOOTH-EXCEPT-W1

SMOOTH-EXCEPT-W2

REASONABLY-SMOOTH-EXCEPT-WEIGHT

CONTINUOUS-EXCEPT-AT-END-POINTS

KNOWN-SINGULARITIES

VECTOR-PROCESSOR

D01AUF

T-PREDICATES: *CONTINUOUS*

VECTOR-PROCESSOR

F-PREDICATES: *CONTINUOUS-EXCEPT-W1*

CONTINUOUS-EXCEPT-W2

REASONABLY-SMOOTH-EXCEPT-WEIGHT

CONTINUOUS-EXCEPT-AT-END-POINTS

KNOWN-SINGULARITIES

D01AHF

T-PREDICATES: *CONTINUOUS-EXCEPT-AT-END-POINTS*

F-PREDICATES: None

D01AJF

T-PREDICATES: None

F-PREDICATES: *VECTOR-PROCESSOR*

CONTINUOUS-EXCEPT-AT-END-POINTS

D01ATF

T-PREDICATES: *VECTOR-PROCESSOR*

F-PREDICATES: *CONTINUOUS-EXCEPT-AT-END-POINTS*

Appendix F

ARC Examples

This appendix sets out a number of examples showing ARC selecting a variety of routines.

```
11: integrate(integrand(x)=log(x),region=[0:1]);
* VECTOR-PROCESSOR is NIL
--> The following routines are being pruned: (D01ATF D01AUF)
--> The following routines have been matched: (D01AJF D01AKF)
* SMOOTH-EXCEPT-W2 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01ANF D01AQF D01ALF)
* CONTINUOUS is NIL
--> The following routines are being pruned: (D01AKF)
--> The following routines have been matched: (D01AQF)
* KNOWN-SINGULARITIES is T
--> The following routines are being pruned: (D01ANF)
--> The following routines have been matched: (D01ALF)
* CONTINUOUS-EXCEPT-AT-END-POINTS is T
--> The following routines are being pruned: (D01AJF D01AQF D01BDF)
--> The following routines have been matched: (D01AHF)
* CONTINUOUS-EXCEPT-W2 is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01APF)
* CONTINUOUS-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01APF)
--> The following routines have been matched: NIL
* REASONABLY-SMOOTH-EXCEPT-WEIGHT is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01ALF)
* REASONABLY-SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01ALF)
*** The recommended routines are :
      (D01ALF D01AHF)
* Now calling routine D01ALF
---> The call terminated unsuccessfully due to an IEEE exception.
* Now calling routine D01AHF
{RELERR,EVALUATIONS,INTEGRAL}
12: integral;
 - 0.999999790039155
```

```
13: integrate(integrand(x)=1/log(sin(log(x))),region=[0:1]);
* VECTOR-PROCESSOR is NIL
--> The following routines are being pruned: (D01ATF D01AUF)
--> The following routines have been matched: (D01AJF D01AKF)
* SMOOTH-EXCEPT-W2 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01ANF D01AQF D01ALF)
* CONTINUOUS is NIL
--> The following routines are being pruned: (D01AKF)
--> The following routines have been matched: (D01AQF)
* KNOWN-SINGULARITIES is NIL
--> The following routines are being pruned: (D01ALF)
--> The following routines have been matched: (D01ANF)
* CONTINUOUS-EXCEPT-AT-END-POINTS is NIL
--> The following routines are being pruned: (D01AHF)
--> The following routines have been matched: (D01AJF D01ANF D01AQF
D01BDF)
* REASONABLY-SMOOTH-EXCEPT-WEIGHT is NIL
--> The following routines are being pruned: (D01ANF)
--> The following routines have been matched: (D01AQF D01APF D01BDF)
* CONTINUOUS-EXCEPT-W2 is NIL
--> The following routines are being pruned: (D01AQF)
--> The following routines have been matched: (D01APF D01BDF)
* CONTINUOUS-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01APF)
--> The following routines have been matched: (D01BDF)
* REASONABLY-SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BDF)
--> The following routines have been matched: NIL
*** The recommended routines are :
      (D01AJF)
* Now calling routine D01AJF
---> The call terminated unsuccessfully due to an IEEE exception.
**** This integration has been unsuccessful.  The following
     routine was tried:
     (D01AJF)
```

```
14: integrate(integrand(x)=sin(x)/x,region=[-1:1]);
* VECTOR-PROCESSOR is NIL
--> The following routines are being pruned: (D01ATF D01AUF)
--> The following routines have been matched: (D01AJF D01AKF)
* SMOOTH-EXCEPT-W2 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01ANF D01AQF D01ALF)
* CONTINUOUS is NIL
--> The following routines are being pruned: (D01AKF)
--> The following routines have been matched: (D01AQF)
* KNOWN-SINGULARITIES is T
--> The following routines are being pruned: (D01ANF)
--> The following routines have been matched: (D01ALF)
* CONTINUOUS-EXCEPT-AT-END-POINTS is NIL
--> The following routines are being pruned: (D01AHF)
--> The following routines have been matched: (D01AJF D01AQF D01BDF)
* CONTINUOUS-EXCEPT-W2 is NIL
--> The following routines are being pruned: (D01AQF)
--> The following routines have been matched: (D01APF D01BDF)
* CONTINUOUS-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01APF)
--> The following routines have been matched: (D01BDF)
* REASONABLY-SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BDF)
--> The following routines have been matched: (D01ALF)
* REASONABLY-SMOOTH-EXCEPT-WEIGHT is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01ALF)
*** The recommended routines are :
      (D01ALF D01AJF)
* Now calling routine D01ALF
{ALIST,BLIST,ELIST,RLIST,ABSERR,INTEGRAL,INTERVALS}
15:  integral;
1.89216614073437
```

```
16: integrate(integrand(x)=sin(20*x)*(x**2-1),region=[0:5]);
* VECTOR-PROCESSOR is NIL
--> The following routines are being pruned: (D01ATF D01AUF)
--> The following routines have been matched: (D01AJF D01AKF)
* SMOOTH-EXCEPT-W2 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01ANF D01AQF D01ALF)
* CONTINUOUS is T
--> The following routines are being pruned: (D01AQF)
--> The following routines have been matched: (D01AKF)
* KNOWN-SINGULARITIES is NIL
--> The following routines are being pruned: (D01ALF)
--> The following routines have been matched: (D01AKF D01ANF)
* CONTINUOUS-EXCEPT-AT-END-POINTS is NIL
--> The following routines are being pruned: (D01AHF)
--> The following routines have been matched: (D01AJF D01AKF D01ANF
D01BDF)
* REASONABLY-SMOOTH-EXCEPT-WEIGHT is T
--> The following routines are being pruned: (D01AKF D01APF D01BDF)
--> The following routines have been matched: (D01ANF)
* CONTINUOUS-EXCEPT-W2 is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01ANF)
* CONTINUOUS-EXCEPT-W1 is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01ANF)
*** The recommended routines are :
      (D01ANF D01AJF)
* Now calling routine D01ANF
{ALIST,BLIST,ELIST,RLIST,ABSERR,TRANSFORM,INTERVALS}
17: transform;
 - 1.09747620805489
```

```
18:  integrate(integrand(x)=x/(x-1),region=[0:2]);
* VECTOR-PROCESSOR is NIL
--> The following routines are being pruned: (D01ATF D01AUF)
--> The following routines have been matched: (D01AJF D01AKF)
* SMOOTH-EXCEPT-W2 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01ANF D01AQF D01ALF)
* CONTINUOUS is NIL
--> The following routines are being pruned: (D01AKF)
--> The following routines have been matched: (D01AQF)
* KNOWN-SINGULARITIES is T
--> The following routines are being pruned: (D01ANF)
--> The following routines have been matched: (D01ALF)
* CONTINUOUS-EXCEPT-AT-END-POINTS is NIL
--> The following routines are being pruned: (D01AHF)
--> The following routines have been matched: (D01AJF D01AQF D01BDF)
* CONTINUOUS-EXCEPT-W2 is T
--> The following routines are being pruned: (D01APF D01BDF)
--> The following routines have been matched: (D01AQF)
* CONTINUOUS-EXCEPT-W1 is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01AQF)
* REASONABLY-SMOOTH-EXCEPT-WEIGHT is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01AQF D01ALF)
* REASONABLY-SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01AQF D01ALF)
*** The recommended routines are :
      (D01AQF D01ALF D01AJF)
* Now calling routine D01AQF
{ALIST,BLIST,ELIST,RLIST,ABSERR,TRANSFORM,INTERVALS}
19: transform;
2.0
```

```
21: off irenalink;

22: integrate(integrand(x)=x*log(x),region=[0:1],vector!-processor);
* VECTOR-PROCESSOR is (VECTOR-PROCESSOR)
--> The following routines are being pruned: (D01AJF D01AKF)
--> The following routines have been matched: (D01ATF D01AUF)
* SMOOTH-EXCEPT-W2 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01ANF D01AQF D01ALF)
* SMOOTH-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01ANF D01AQF D01ALF)
* SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01ANF D01AQF D01ALF)
* CONTINUOUS is T
--> The following routines are being pruned: (D01AQF)
--> The following routines have been matched: (D01AUF)
* KNOWN-SINGULARITIES is NIL
--> The following routines are being pruned: (D01ALF)
--> The following routines have been matched: (D01AUF D01ANF)
* CONTINUOUS-EXCEPT-AT-END-POINTS is NIL
--> The following routines are being pruned: (D01AHF)
--> The following routines have been matched: (D01ATF D01AUF D01ANF
D01BDF)
* REASONABLY-SMOOTH-EXCEPT-WEIGHT is NIL
--> The following routines are being pruned: (D01ANF)
--> The following routines have been matched: (D01AUF D01APF D01BDF)
* CONTINUOUS-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01APF)
--> The following routines have been matched: (D01AUF D01BDF)
* CONTINUOUS-EXCEPT-W2 is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01AUF D01BDF)
* REASONABLY-SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BDF)
--> The following routines have been matched: NIL

{D01AUF,D01ATF}
```

```
23: integrate(integrand(x)=1/cos(log(x)),region=[0:pi],
23:           vector!-processor);
* VECTOR-PROCESSOR is (VECTOR-PROCESSOR)
--> The following routines are being pruned: (D01AJF D01AKF)
--> The following routines have been matched: (D01ATF D01AUF)
* SMOOTH-EXCEPT-W2 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01ANF D01AQF D01ALF)
* SMOOTH-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01ANF D01AQF D01ALF)
* SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01ANF D01AQF D01ALF)
* CONTINUOUS is NIL
--> The following routines are being pruned: (D01AUF)
--> The following routines have been matched: (D01AQF)
* KNOWN-SINGULARITIES is NIL
--> The following routines are being pruned: (D01ALF)
--> The following routines have been matched: (D01ANF)
* CONTINUOUS-EXCEPT-AT-END-POINTS is NIL
--> The following routines are being pruned: (D01AHF)
--> The following routines have been matched: (D01ATF D01ANF D01AQF
D01BDF)
* REASONABLY-SMOOTH-EXCEPT-WEIGHT is NIL
--> The following routines are being pruned: (D01ANF)
--> The following routines have been matched: (D01AQF D01APF D01BDF)
* CONTINUOUS-EXCEPT-W2 is NIL
--> The following routines are being pruned: (D01AQF)
--> The following routines have been matched: (D01APF D01BDF)
* CONTINUOUS-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01APF)
--> The following routines have been matched: (D01BDF)
* REASONABLY-SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BDF)
--> The following routines have been matched: NIL

{D01ATF}
```

```
26: integrate(integrand(x)=x^2*log(x),region=[0:1]);
* VECTOR-PROCESSOR is NIL
--> The following routines are being pruned: (D01ATF D01AUF)
--> The following routines have been matched: (D01AJF D01AKF)
* SMOOTH-EXCEPT-W2 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-INTEGRAND is T
--> The following routines are being pruned: (D01ANF D01AQF D01ALF)
--> The following routines have been matched: (D01BAF D01ARF)
* CONTINUOUS-EXCEPT-AT-END-POINTS is NIL
--> The following routines are being pruned: (D01AHF)
--> The following routines have been matched: (D01AJF D01AKF D01BDF
D01BAF D01ARF)
* CONTINUOUS-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01APF)
--> The following routines have been matched: (D01AKF D01BDF D01BAF
D01ARF)
* KNOWN-SINGULARITIES is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01AKF)
* CONTINUOUS is T
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01AKF)
* REASONABLY-SMOOTH-INTEGRAND is T
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01BDF)
* REASONABLY-SMOOTH-EXCEPT-WEIGHT is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01AKF D01BDF D01BAF
D01ARF)
* CONTINUOUS-EXCEPT-W2 is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01AKF D01BDF D01BAF
D01ARF)
*** The recommended routines are :
      (D01AKF D01BDF D01BAF D01ARF D01AJF)
* Now calling routine D01AKF
{ALIST,BLIST,ELIST,RLIST,ABSERR,INTEGRAL,INTERVALS}
27: integral;
 - 0.11111111111105
```

```
28: integrate(integrand(x)=abs(x-1/2)^3*sin(x),region=[0:1]);
* VECTOR-PROCESSOR is NIL
--> The following routines are being pruned: (D01ATF D01AUF)
--> The following routines have been matched: (D01AJF D01AKF)
* SMOOTH-EXCEPT-W2 is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: (D01AKF D01ANF D01AQF
D01ALF)
* SMOOTH-EXCEPT-W1 is T
--> The following routines are being pruned: (D01AKF D01ANF D01AQF
D01ALF)
--> The following routines have been matched: (D01BAF D01ARF)
* CONTINUOUS-EXCEPT-AT-END-POINTS is NIL
--> The following routines are being pruned: (D01AHF)
--> The following routines have been matched: (D01AJF D01BDF D01BAF
D01BAF D01ARF D01ARF)
* CONTINUOUS-EXCEPT-W1 is NIL
--> The following routines are being pruned: (D01APF)
--> The following routines have been matched: (D01BDF D01BAF D01BAF
D01ARF D01ARF)
* SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BAF D01ARF)
--> The following routines have been matched: NIL
* REASONABLY-SMOOTH-INTEGRAND is NIL
--> The following routines are being pruned: (D01BDF)
--> The following routines have been matched: NIL
* REASONABLY-SMOOTH-EXCEPT-WEIGHT is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01BAF D01ARF)
* CONTINUOUS-EXCEPT-W2 is NIL
--> The following routines are being pruned: NIL
--> The following routines have been matched: (D01BAF D01ARF)
*** The recommended routines are :
      (D01BAF D01ARF D01AJF)
* Now calling routine D01BAF
{INTEGRAL}
29: integral;
0.00171911498967147
```

Bibliography

[ANSI 1978] ANSI. American National Standard Programming Language Fortran. Technical Report ANS X3.9, American National Standards Institute, 1978.

[ANSI 1989] ANSI. Fortran 8x. Technical Report X3J3/S8.112, American National Standards Institute, June 1989.

[Barbier *et al.*] C. Barbier, P. Clark, P. Bettess, and J. Bettess. Automatic generation of shape functions for finite element analysis using REDUCE. Submitted to *International Journal for Numerical Methods in Engineering.*

[Barnes & Padget 1990] A. Barnes and J. Padget. Univariate Power Series Expansions In REDUCE. In *Proceedings of ISSAC '90*, 1990.

[Barton *et al.* 1971] D. R. Barton, I. M. Willers, and R. V. M. Zahar. The Automatic Solution of Systems of Ordinary Differential Equations by the Method of Taylor Series. *Computer Journal*, 14:243–248, 1971.

[Bennett *et al.* 1988] J. P. Bennett, J. H. Davenport, and H. M. Sauro. Solution of Some Equations in Biochemistry. Technical Report 88-12, University of Bath, June 1988.

[Boisvert *et al.* 1985] Ronald F. Boisvert, Sally E. Howe, and David K. Kahaner. GAMS: A Framework for the Management of Scientific Software. *ACM Transactions on Mathematical Software*, 11(4):313–355, December 1985.

[Bradford *et al.* 1986] R. J. Bradford, A. C. Hearn, J. A. Padget, and E. Schrüfer. Enlarging the REDUCE Domain of Computation. In *Proceedings of SYMSAC '86*, pages 100–106, 1986.

[Broughan 1986] Kevin A. Broughan. A Symbolic Numeric Interface for the NAG Library. *The NAG Newsletter*, (2):16–24, 1986.

[Broughan 1987] Kevin A. Broughan. Naglink — A Working Symbolic / Numeric Interface. In B. Ford and F. Chatelin, editors, *Problem Solving Environments for Scientific Computing*, pages 343–347. North Holland, 1987.

[Davenport 1985] James H. Davenport. The Symbolic and Numeric Manipulation of Integrals. In *Lecture Notes in Computer Science*, volume **235**, pages 168–180. Springer-Verlag, 1985.

[Davenport *et al.* 1988] James H. Davenport, Yves Siret, and E. Tournier. *Computer Algebra.* Academic Press, London, 1988.

[Dewar & Richardson 1990] Michael C. Dewar and Michael G. Richardson. Reconciling Symbolic and Numeric Computation in a Practical Setting. In *Proceedings of DISCO '90*, pages 195–204. Springer-Verlag, 1990.

[Dewar 1989] Michael C. Dewar. IRENA —An Integrated Symbolic and Numerical Computation Environment. In *Proceedings of ISSAC 1989*, pages 171–179. ACM, 1989.

[Fisher 1990a] D. L. Fisher. Applications of Computer Algebra to Enzyme Analysis. Technical Report 90-32, University of Bath, February 1990.

[Fisher 1990b] D. L. Fisher. Novel Computer Techniques in Enzyme Kinetics. Technical Report 90-41, University of Bath, August 1990.

[Fitch 1979] J. P. Fitch. The application of symbolic algebra to physics — a case of creeping flow. In *Proceedings of EUROSAM '79*, pages 30–41. Springer-Verlag, 1979.

[Fitch 1985] J. P. Fitch. Solving Algebraic Problems with REDUCE. *Journal of Symbolic Computation*, 1(1):211–227, 1985.

[Fitch 1990] J. P. Fitch. The algebraic - numeric interface. *Computer Physics Communications*, (61):22–33, 1990.

[Gaffney & Wooten 1983] P. W. Gaffney and J. W. Wooten. NITPACK: An Interactive Tree Package. *ACM Transactions on Mathematical Software*, 9(4):395–417, December 1983.

[Galway *et al.* 1987] W. Galway, M. L. Griss, B. Morrison, and B. Othmer. *The PSL 3.4 User's Manual.* University of Utah, 1987.

[Gates & Wang 1984] Barbara L. Gates and Paul S. Wang. A LISP-based RATFOR Code Generator. In *Proceedings of the 1984 MACSYMA User's Conference*, 1984.

[Gates 1985] Barbara L. Gates. GENTRAN: An Automatic Code Generation Facility for REDUCE. *ACM SIGSAM Bulletin*, 19(3):24–42, August 1985.

[Gates 1986] Barbara L. Gates. A Numerical Code Generation Facility for REDUCE. In *Proceedings of SYMSAC 1986*, pages 94–99. ACM, 1986.

[Gates 1987] Barbara L. Gates. *The GENTRAN User's Manual: REDUCE Version.* The RAND Corporation, 1987.

[Gomez 1990] Claude Gomez. MACROFORT: A FORTRAN Code Generator for Maple. Technical Report 119, Institut National de Recherche en Informatique et en Automatique, May 1990.

[Hazel & O'Donohue 1980] P. Hazel and M. R. O'Donohue. HELP Numerical: The Cambridge interactive documentation system for numerical methods. In *Production and Assessment of Numerical Software*, pages 367–382. Academic Press, 1980.

[Hearn 1987] A. H. Hearn. *The REDUCE User's Manual.* The RAND Corporation, 1987.

[Hulshof 1983] B. J. A. Hulshof. *COMPRESS.* The RAND Corporation, 1983.

[Kameny 1990] Stanley L. Kameny. *The REDUCE Root Finding Package.* The RAND Corporation, 1990.

[Kant *et al.* 1990] E. Kant, F. Daube, B. MacGregor, and J. Wald. MathCode: A Code Generation Package for Mathematica. Technical report, Schlumberger Technologies Corporation, September 1990.

[Kornerup & Matula 1979] P. Kornerup and D. W. Matula. Approximate Rational Arithmetic Systems: Analysis of Recovery of Simple Fractions During Expression Evaluation. In *Proceedings of EUROSAM '79*, pages 383–397. Springer-Verlag, 1979.

[McIsaac 1990] Kevin McIsaac. *PM — a REDUCE Pattern Matcher.* The RAND Corporation, 1990.

[Mutrie *et al.* 1987] Mark P. W. Mutrie, Bruce W. Char, and Richard H. Bartels. Expression Optimization Using High-Level Knowledge. In *Proceedings of EUROCAL '87*, pages 64–70. Springer-Verlag, 1987.

[NAG LTD. 1989] NAG LTD. The KASTLE System. Technical report, The FOCUS Consortium, April 1989.

[NAG LTD. 1990] NAG LTD. *The NAG Fortran Library Manual — Mark 14*, 1990.

[Ng 1979] Edward W. Ng. Symbolic–Numeric Interface: A Review. In *Proceedings of EUROSAM '79*, pages 330–345. Springer-Verlag, 1979.

[Punjani & Broughan 1990] Minaz Punjani and Kevin A. Broughan. SENAC — Computer based Algebra and NAG. *ULCC News*, pages 12–15, October 1990.

[Richardson 1988] Michael G. Richardson. Suggested Representation of Matrices in IRENA. Private Communication, July 1988.

[Sasaki 1979] T. Sasaki. An Arbitrary Precision Real Arithmetic Package in REDUCE. In *Proceedings of EUROSAM '79*, pages 358–368. Springer-Verlag, 1979.

[Schou & Broughan 1989] Wayne C. Schou and Kevin A. Broughan. The Risch Algorithms of MACSYMA and SENAC. *ACM SIGSAM Bulletin*, **23**(3):19–22, 1989.

[Schulze & Cryer 1986] Klaus Schulze and Colin W. Cryer. NAXPERT: A Prototype Expert System for Numerical Analysis. Technical Report 7/86 1, Institut für Numerische und instrumentelle Mathematik, November 1986.

[Schulze & Cryer 1988] Klaus Schulze and Colin W. Cryer. NAXPERT: A Prototype Expert System for Numerical Software. *SIAM Journal of Scientific and Statistical Computation*, **9**(3), May 1988.

[van Hulzen 1984] J. A. van Hulzen. Code optimization by symbolic processing. In *NGI-SION Symposium*, pages 194–203, 1984.

[van Hulzen *et al.* 1989] J. A. van Hulzen, B. J. A. Hulshof, B. L. Gates, and M. C. van Heerwaarden. A Code Optimization Package for REDUCE. In *Proceedings of ISSAC 1989*, pages 163–170. ACM, 1989.

[Wang 1985] Paul S. Wang. Taking Advantage of Symmetry in the Automatic Generation of Numerical Programs for Finite Element Analysis. In *Proceedings of EUROCAL '85*, pages 572–582. Springer-Verlag, 1985.

[Wang 1986] Paul S. Wang. FINGER: A Symbolic System for Automatic Generation of Numerical Programs in Finite Element Analysis. *Journal of Symbolic Computation*, **2**:305–316, 1986.

[Weerawarana & Wang 1989] Sanjiva Weerawarana and Paul S. Wang. GENCRAY: A Portable Code Generator for Cray Fortran. In *Proceedings of ISSAC 1989*, pages 186–191. ACM, 1989.

Index